Oracle® TimesTen In-Memory Database

PL/SQL Packages Reference

Release 11.2.1

E14000-04

March 2012

Oracle TimesTen In-Memory Database PL/SQL Packages Reference, Release 11.2.1

E14000-04

Copyright © 1996, 2012, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

4 DBMS_PREPROCESSOR

5 DBMS_RANDOM

6 DBMS_SQL

7 DBMS_UTILITY

8 TT_DB_VERSION

9 UTL_FILE

10 UTL_IDENT

11 UTL_RAW

12 UTL_RECOMP

Index

Preface

Oracle TimesTen In-Memory Database is a memory-optimized relational database. Deployed in the application tier, TimesTen operates on databases that fit entirely in physical memory using standard SQL interfaces. High availability for the in-memory database is provided through real-time transactional replication.

TimesTen supports a variety of programming interfaces, including ODBC (Open Database Connectivity), OCI (Oracle Call Interface), Oracle Pro*C/C++ (precompiler for embedded SQL and PL/SQL instructions in C or C++ code), JDBC (Java Database Connectivity), and PL/SQL (Oracle procedural language extension for SQL).

This Preface contains these topics:

- Audience
- Related Documents
- Conventions
- Documentation Accessibility
- Technical support

Audience

Oracle TimesTen In-Memory Database PL/SQL Packages Reference is a reference for programmers, systems analysts, project managers, and others interested in developing database applications using PL/SQL. This manual assumes a working knowledge of application programming and familiarity with SQL and PL/SQL to access information in relational database systems.

Related Documents

TimesTen documentation is available on the product distribution media and on the Oracle Technology Network:

`http://www.oracle.com/technetwork/database/timesten/documentation/`

Oracle TimesTen In-Memory Database PL/SQL Developer's Guide is especially relevant.

Oracle documentation is also available on the Oracle Technology network. This may be especially useful for Oracle features that TimesTen supports but does not attempt to fully document:

`http://www.oracle.com/technetwork/database/enterprise-edition/documentation/`

In particular, the following Oracle documents may be of interest.

- *Oracle Database PL/SQL Language Reference*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Reference*

In addition, numerous third-party documents are available that describe PL/SQL in detail.

Conventions

TimesTen supports multiple platforms. Unless otherwise indicated, the information in this guide applies to all supported platforms. The term Windows applies to all supported Windows platforms. The term UNIX applies to all supported UNIX and Linux platforms. Refer to the "Platforms" section in *Oracle TimesTen In-Memory Database Release Notes* for specific platform versions supported by TimesTen.

Note: In TimesTen documentation, the terms "data store" and "database" are equivalent. Both terms refer to the TimesTen database unless otherwise noted.

This document uses the following text conventions:

Convention	Meaning
italic	Italic type indicates terms defined in text, book titles, or emphasis.
`monospace`	Monospace type indicates commands, URLs, procedure and function names, package names, attribute names, directory names, file names, text that appears on the screen, or text that you enter.
`italic monospace`	Italic monospace type indicates a variable in a code example that you must replace, as in the following example. `Driver=install_dir/lib/libtten.sl` This means you should replace *`install_dir`* with the path of your TimesTen installation directory.
[ ]	Square brackets indicate that an item in a command line is optional.
{ }	Curly braces indicated that you must choose one of the items separated by a vertical bar (|) in a command line.
|	A vertical bar (or pipe) separates alternative arguments.
. . .	An ellipsis (. . .) after an argument indicates that you may use more than one argument on a single command line.
%	The percent sign indicates the UNIX shell prompt.

TimesTen documentation uses these variables to identify path, file and user names:

Convention	Meaning
`install_dir`	The path that represents the directory where TimesTen is installed.
`TTinstance`	The instance name for your specific installation of TimesTen. Each installation of TimesTen must be identified at install time with a unique instance name. This name appears in the install path.

Convention	Meaning
`bits` or *`bb`*	Two digits, either 32 or 64, that represent either the 32-bit or 64-bit operating system.
`release` or *`rr`*	Numbers that represent a major TimesTen release, with or without dots. For example, 1121 or 11.2.1 represents TimesTen Release 11.2.1.
`DSN`	The data source name (for the TimesTen database).

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Technical support

For information about obtaining technical support for TimesTen products, go to the following Web address:

`http://www.oracle.com/support/contact.html`

1

Introduction to TimesTen-Supplied PL/SQL Packages and Types

A set of PL/SQL packages is installed when you enable PL/SQL for the TimesTen In-Memory Database. These packages extend database functionality and allow PL/SQL access to SQL features. To display the list of packages provided with TimesTen, use the system view `ALL_PROCEDURES` for objects owned by `SYS`. The following example shows this. As with other `ALL_*` system views, all users have `SELECT` privilege for the `ALL_PROCEDURES` system view.

```
Command> select distinct object_name from all_procedures where owner='SYS';
< DBMS_LOCK >
< DBMS_OUTPUT >
< DBMS_PREPROCESSOR >
< DBMS_RANDOM >
< DBMS_SQL >
< DBMS_STANDARD >
< DBMS_SYS_ERROR >
< DBMS_UTILITY >
< PLITBLM >
< STANDARD >
< SYS_STUB_FOR_PURITY_ANALYSIS >
< UTL_FILE >
< UTL_RAW >
< UTL_RECOMP >
< TT_DB_VERSION >
< UTL_IDENT >
16 rows found.
```

This is the list of TimesTen-supplied packages currently installed in the database. It includes internal-use packages and public packages. This manual documents only the public packages, as summarized in "Summary of TimesTen-supplied PL/SQL packages" on page 1-8.

Notes: The following packages are for internal use only and are not documented in this manual:

- `STANDARD`, `PLITBLM`, and `DBMS_STANDARD`: Programs defined in these packages are part of the PL/SQL language.
- `DBMS_SYS_ERROR`: Defines private PL/SQL subprograms for system error messages arising from `DBMS*` routines.
- `SYS_STUB_FOR_PURITY_ANALYSIS`: Required for the creation of top-level subprograms.

This chapter contains these topics:

- Package overview
- Summary of TimesTen-supplied PL/SQL packages

Package overview

A *package* is an encapsulated collection of related program objects stored together in the database. Program objects are procedures, functions, variables, constants, cursors, and exceptions.

This section covers the following topics:

- Package components
- Using TimesTen-supplied packages
- Referencing package contents
- Running package examples

Package components

PL/SQL packages have two parts, the specification and the body, although sometimes the body is unnecessary. The specification is the interface to your application. It declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the specification.

Unlike subprograms, packages cannot be called, parameterized, or nested. However, the formats of a package and a subprogram are similar:

```
CREATE PACKAGE name AS  -- specification (visible part)
   -- public type and item declarations
   -- subprogram specifications
END [name];

CREATE PACKAGE BODY name AS  -- body (hidden part)
   -- private type and item declarations
   -- subprogram bodies
[BEGIN
   -- initialization statements]
END [name];
```

The specification holds public declarations that are visible to your application. The body holds implementation details and private declarations that are hidden from your application. You can debug, enhance, or replace a package body without changing the specification. You can change a package body without recompiling calling programs because the implementation details in the body are hidden from your application.

Using TimesTen-supplied packages

TimesTen-supplied packages are automatically installed when the database is created.

All users have `EXECUTE` privilege for packages described in this document, other than for `UTL_RECOMP` and `UTL_FILE`, as noted in those chapters.

To select from a view defined with a PL/SQL function, you must have `SELECT` privileges on the view. No separate `EXECUTE` privileges are needed to select from the view. Instructions on special requirements for packages are documented in the individual chapters.

Note: In TimesTen, running as the instance administrator is comparable to running as the Oracle user `SYSDBA`. Running as the `ADMIN` user is comparable to running as the Oracle user `DBA`.

Referencing package contents

To reference the types, items, and subprograms declared in a package specification, use "dot" notation. For example:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
```

Running package examples

In order to see the output from the package examples in this document, first execute the following command in `ttIsql`:

```
Command> set serveroutput on
```

Summary of TimesTen-supplied PL/SQL packages

Table 1–1 lists the PL/SQL server packages supplied with TimesTen for public use. These packages run as the invoking user, rather than the package owner.

Caution:

- The procedures and functions provided in these packages and their external interfaces are reserved by Oracle and are subject to change.
- Modifying supplied packages can cause internal errors and database security violations. Do not modify supplied packages.

Table 1–1 Summary of TimesTen-supplied PL/SQL packages

Package Name	Description
`DBMS_LOCK`	Provides an interface to Lock Management services. TimesTen supports only the `SLEEP` procedure, to suspend the session for a specified duration.
`DBMS_OUTPUT`	Enables you to send messages from stored procedures and packages.
`DBMS_PREPROCESSOR`	Provides an interface to print or retrieve the source text of a PL/SQL unit in its post-processed form.
`DBMS_RANDOM`	Provides a built-in random number generator.
`DBMS_SQL`	Lets you use dynamic SQL to access the database.
`DBMS_UTILITY`	Provides various utility routines.
`TT_DB_VERSION`	Indicates the TimesTen major and minor version numbers.
`UTL_FILE`	Enables your PL/SQL programs to read and write operating system text files and provides a restricted version of standard operating system stream file I/O.
`UTL_IDENT`	Indicates in which database or client PL/SQL is running, such as TimesTen versus Oracle, and server versus client. (Each database or client running PL/SQL has its own copy of this package.)
`UTL_RAW`	Provides SQL functions for manipulating `RAW` data types.
`UTL_RECOMP`	Recompiles invalid PL/SQL modules.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

2

DBMS_LOCK

The `DBMS_LOCK` package provides an interface to Lock Management services.

In the current release, TimesTen supports only the `SLEEP` subprogram.

This chapter contains the following topics:

- Using DBMS_LOCK
- Summary of DBMS_LOCK subprograms

Using DBMS_LOCK

TimesTen currently implements only the `SLEEP` subprogram, used to suspend the session for a specified duration.

Summary of DBMS_LOCK subprograms

In the current release, TimesTen supports only the `SLEEP` subprogram.

Table 2–1 DBMS_LOCK package subprograms

Subprogram	Description
SLEEP procedure	Suspends the session for a specified duration.

SLEEP procedure

This procedure suspends the session for a specified duration.

Syntax

```
DBMS_LOCK.SLEEP (
   seconds  IN NUMBER);
```

Parameters

Table 2–2 SLEEP procedure parameters

Parameter	Description
seconds	Amount of time, in seconds, to suspend the session. The smallest increment is a hundredth of a second.

Usage notes

- The actual sleep time may be somewhat longer than specified, depending on system activity.
- If the `PLSQL_TIMEOUT` general connection attribute is set to a positive value that is less than this sleep time, the timeout will take effect first. Be sure that either the sleep value is less than the timeout value, or `PLSQL_TIMEOUT=0` (no timeout). See "PL/SQL connection attributes" in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* for information about `PLSQL_TIMEOUT`.

Examples

```
DBMS_LOCK.SLEEP(1.95);
```

3

DBMS_OUTPUT

The `DBMS_OUTPUT` package enables you to send messages from stored procedures and packages. The package is especially useful for displaying PL/SQL debugging information.

This chapter contains the following topics:

- Using DBMS_OUTPUT
  - Overview
  - Operational notes
  - Rules and limits
  - Exceptions
  - Examples
- Data structures
  - Table types
- Summary of DBMS_OUTPUT subprograms

Using DBMS_OUTPUT

This section contains topics which relate to using the `DBMS_OUTPUT` package.

- Overview
- Operational notes
- Exceptions
- Rules and limits
- Example

Overview

The PUT procedure and PUT_LINE procedure in this package enable you to place information in a buffer that can be read by another procedure or package. In a separate PL/SQL procedure or anonymous block, you can display the buffered information by calling the GET_LINE procedure and GET_LINES procedure.

If the package is disabled, all calls to subprograms are ignored. In this way, you can design your application so that subprograms are available only when a client can process the information.

Operational notes

- If you do not call `GET_LINE`, or if you do not display the messages on your screen in `ttIsql`, the buffered messages are ignored.
- The `ttIsql` utility calls `GET_LINES` after issuing a SQL statement or anonymous PL/SQL calls.
- Typing `SET SERVEROUTPUT ON` in `ttIsql` has the same effect as the following:

  ```
  DBMS_OUTPUT.ENABLE (buffer_size => NULL);
  ```

  There is no limit on the output.
- You should generally avoid having application code invoke either the DISABLE procedure or ENABLE procedure because this could subvert the attempt of an external tool like `ttIsql` to control whether to display output.

Note: Messages sent using `DBMS_OUTPUT` are not actually sent until the sending subprogram completes. There is no mechanism to flush output during the execution of a procedure.

Rules and limits

- The maximum line size is 32767 bytes.
- The default buffer size is 20000 bytes. The minimum size is 2000 bytes and the maximum is unlimited.

Exceptions

`DBMS_OUTPUT` subprograms raise the application error `ORA-20000`, and the output procedures can return the following errors:

Table 3–1 DBMS_OUTPUT exceptions

Exception	Description
`ORU-10027`	Buffer overflow.
`ORU-10028`	Line length overflow.

Example

Example: Debugging stored procedures

The `DBMS_OUTPUT` package is commonly used to debug stored procedures or functions.

This function queries the `employees` table of the `HR` schema and returns the total salary for a specified department. The function includes calls to the `PUT_LINE` procedure:

```
CREATE OR REPLACE FUNCTION dept_salary (dnum NUMBER) RETURN NUMBER IS
   CURSOR emp_cursor IS
   select salary, commission_pct from employees where department_id = dnum;
   total_wages NUMBER(11, 2) := 0;
   counter NUMBER(10) := 1;
BEGIN
   FOR emp_record IN emp_cursor LOOP
       emp_record.commission_pct := NVL(emp_record.commission_pct, 0);
       total_wages := total_wages + emp_record.salary
                   + emp_record.commission_pct;
       DBMS_OUTPUT.PUT_LINE('Loop number = ' || counter ||
          '; Wages = '|| TO_CHAR(total_wages)); /* Debug line */
       counter := counter + 1; /* Increment debug counter */
   END LOOP;
   /* Debug line */
   DBMS_OUTPUT.PUT_LINE('Total wages = ' ||
   TO_CHAR(total_wages));
   RETURN total_wages;
END;
/
```

Assume the user executes the following statements in `ttIsql`:

```
Command> SET SERVEROUTPUT ON
Command> VARIABLE salary NUMBER;
Command> EXECUTE :salary := dept_salary(20);
```

The user would then see output such as the following:

```
Loop number = 1; Wages = 13000
Loop number = 2; Wages = 19000
Total wages = 19000

PL/SQL procedure successfully executed.
```

Data structures

The `DBMS_OUTPUT` package declares two table types for use with the GET_LINES procedure.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

Table types

CHARARR table type

DBMSOUTPUT_LINESARRAY table type

CHARARR table type

This package type is to be used with the GET_LINES procedure to obtain text submitted through the PUT procedure and PUT_LINE procedure.

Syntax

```
TYPE CHARARR IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

DBMSOUTPUT_LINESARRAY table type

This package type is to be used with the GET_LINES procedure to obtain text submitted through the PUT procedure and PUT_LINE procedure.

Syntax

```
TYPE DBMSOUTPUT_LINESARRAY IS
     VARRAY(2147483647) OF VARCHAR2(32767);
```

Summary of DBMS_OUTPUT subprograms

Table 3–2 DBMS_OUTPUT package subprograms

Subprogram	Description
DISABLE procedure	Disables message output.
ENABLE procedure	Enables message output.
GET_LINE procedure	Retrieves one line from buffer.
GET_LINES procedure	Retrieves an array of lines from buffer.
NEW_LINE procedure	Terminates a line created with `PUT`.
PUT procedure	Places a line in the buffer.
PUT_LINE procedure	Places partial line in buffer.

DISABLE procedure

This procedure disables calls to `PUT`, `PUT_LINE`, `NEW_LINE`, `GET_LINE`, and `GET_LINES`, and purges the buffer of any remaining information.

As with the ENABLE procedure, you do not need to call this procedure if you are using the `SET SERVEROUTPUT ON` setting from `ttIsql`.

Syntax

```
DBMS_OUTPUT.DISABLE;
```

ENABLE procedure

This procedure enables calls to `PUT`, `PUT_LINE`, `NEW_LINE`, `GET_LINE`, and `GET_LINES`. Calls to these procedures are ignored if the `DBMS_OUTPUT` package is not activated.

Syntax

```
DBMS_OUTPUT.ENABLE (
   buffer_size IN INTEGER DEFAULT 20000);
```

Parameters

Table 3–3 ENABLE procedure parameters

Parameter	**Description**
`buffer_size`	Upper limit, in bytes, for the amount of buffered information. Setting *`buffer_size`* to `NULL` specifies that there should be no limit.

Usage notes

- It is not necessary to call this procedure when you use `SET SERVEROUTPUT ON` from `ttIsql`. It is called automatically (with `NULL` value for *`buffer_size`* in the current release).
- If there are multiple calls to `ENABLE`, then *`buffer_size`* is the last of the values specified. The maximum size is 1,000,000 and the minimum is 2000 when the user specifies *`buffer_size`* (`NOT NULL`).
- `NULL` is expected to be the usual choice. The default is 20000 for backward compatibility with earlier database versions that did not support unlimited buffering.

GET_LINE procedure

This procedure retrieves a single line of buffered information.

Syntax

```
DBMS_OUTPUT.GET_LINE (
   line    OUT VARCHAR2,
   status  OUT INTEGER);
```

Parameters

Table 3–4 GET_LINE procedure parameters

Parameter	Description
`line`	Returns a single line of buffered information, excluding a final newline character. You should declare this parameter as `VARCHAR2(32767)` to avoid the risk of "`ORA-06502: PL/SQL: numeric or value error: character string buffer too small`".
`status`	If the call completes successfully, then the status returns as 0. If there are no more lines in the buffer, then the status is 1.

Usage notes

- You can choose to retrieve from the buffer a single line or an array of lines. Call `GET_LINE` to retrieve a single line of buffered information. To reduce the number of calls to the server, call `GET_LINES` to retrieve an array of lines from the buffer.
- You can choose to automatically display this information if you are using `ttIsql` by using the special `SET SERVEROUTPUT ON` command.
- After calling `GET_LINE` or `GET_LINES`, any lines not retrieved before the next call to `PUT`, `PUT_LINE`, or `NEW_LINE` are discarded to avoid confusing them with the next message.

GET_LINES procedure

This procedure retrieves an array of lines from the buffer.

Syntax

```
DBMS_OUTPUT.GET_LINES (
   lines       OUT     DBMS_OUTPUT.CHARARR,
   numlines    IN OUT  INTEGER);

DBMS_OUTPUT.GET_LINES (
   lines       OUT     DBMS_OUTPUT.DBMSOUTPUT_LINESARRAY,
   numlines    IN OUT INTEGER);
```

Parameters

Table 3–5 GET_LINES procedure parameters

Parameter	Description
`lines`	Returns an array of lines of buffered information. The maximum length of each line in the array is 32767 bytes. It is recommended that you use the VARRAY overload version in a 3GL host program to execute the procedure from a PL/SQL anonymous block.
`numlines`	Number of lines you want to retrieve from the buffer. After retrieving the specified number of lines, the procedure returns the number of lines actually retrieved. If this number is less than the number of lines requested, then there are no more lines in the buffer.

Usage notes

- You can choose to retrieve from the buffer a single line or an array of lines. Call `GET_LINE` to retrieve a single line of buffered information. To reduce the number of trips to the server, call `GET_LINES` to retrieve an array of lines from the buffer.
- You can choose to automatically display this information if you are using `ttIsql` by using the special `SET SERVEROUTPUT ON` command.
- After `GET_LINE` or `GET_LINES` is called, any lines not retrieved before the next call to `PUT`, `PUT_LINE`, or `NEW_LINE` are discarded to avoid confusing them with the next message.

NEW_LINE procedure

This procedure puts an end-of-line marker. The GET_LINE procedure and the GET_LINES procedure return "lines" as delimited by "newlines". Every call to the PUT_LINE procedure or to `NEW_LINE` generates a line that is returned by `GET_LINE` or `GET_LINES`.

Syntax

```
DBMS_OUTPUT.NEW_LINE;
```

PUT procedure

This procedure places a partial line in the buffer.

> **Note:** The `PUT` version that takes a `NUMBER` input is obsolete. It is supported for legacy reasons only.

Syntax

```
DBMS_OUTPUT.PUT (
    a IN VARCHAR2);
```

Parameters

Table 3–6 PUT procedure parameters

Parameter	Description
a	Item to buffer.

Usage notes

- You can build a line of information piece by piece by making multiple calls to `PUT`, or place an entire line of information into the buffer by calling `PUT_LINE`.
- When you call `PUT_LINE`, the item you specify is automatically followed by an end-of-line marker. If you make calls to `PUT` to build a line, you must add your own end-of-line marker by calling `NEW_LINE`. `GET_LINE` and `GET_LINES` do not return lines that have not been terminated with a newline character.
- If your lines exceed the line limit, you receive an error message.
- Output that you create using `PUT` or `PUT_LINE` is buffered. The output cannot be retrieved until the PL/SQL program unit from which it was buffered returns to its caller.

Exceptions

Table 3–7 PUT procedure exceptions

Exception	Description
`ORA-20000, ORU-10027`	Buffer overflow, according to the *`buffer_size`* limit specified in the ENABLE procedure call.
`ORA-20000, ORU-10028`	Line length overflow, limit of 32767 bytes for each line.

PUT_LINE procedure

This procedure places a line in the buffer.

> **Note:** The `PUT_LINE` version that takes a `NUMBER` input is obsolete. It is supported for legacy reasons only.

Syntax

```
DBMS_OUTPUT.PUT_LINE (
   a IN VARCHAR2);
```

Parameters

Table 3–8 PUT_LINE procedure parameters

Parameter	Description
a	Item to buffer.

Usage notes

- You can build a line of information piece by piece by making multiple calls to `PUT`, or place an entire line of information into the buffer by calling `PUT_LINE`.
- When you call `PUT_LINE`, the item you specify is automatically followed by an end-of-line marker. If you make calls to `PUT` to build a line, then you must add your own end-of-line marker by calling `NEW_LINE`. `GET_LINE` and `GET_LINES` do not return lines that have not been terminated with a newline character.
- If your lines exceeds the line limit, you receive an error message.
- Output that you create using `PUT` or `PUT_LINE` is buffered. The output cannot be retrieved until the PL/SQL program unit from which it was buffered returns to its caller.

Exceptions

Table 3–9 PUT_LINE procedure exceptions

Exception	Description
`ORA-20000, ORU-10027`	Buffer overflow, according to the *`buffer_size`* limit specified in the ENABLE procedure call.
`ORA-20000, ORU-10028`	Line length overflow, limit of 32767 bytes for each line.

4

DBMS_PREPROCESSOR

The `DBMS_PREPROCESSOR` package provides an interface to print or retrieve the source text of a PL/SQL unit in its post-processed form.

This package contains the following topics:

- Using DBMS_PREPROCESSOR
  - Overview
  - Operating notes
- Data structures
  - Table types
- Summary of DBMS_PREPROCESSOR subprograms

Using DBMS_PREPROCESSOR

- Overview
- Operating notes

Overview

There are three styles of subprograms:

1. Subprograms that take a schema name, a unit type name, and the unit name.
2. Subprograms that take a `VARCHAR2` string which contains the source text of an arbitrary PL/SQL compilation unit.
3. Subprograms that take a `VARCHAR2` associative array (index-by table) which contains the segmented source text of an arbitrary PL/SQL compilation unit.

Subprograms of the first style are used to print or retrieve the post-processed source text of a stored PL/SQL unit. The user must have the privileges necessary to view the original source text of this unit. The user must also specify the schema in which the unit is defined, the type of the unit, and the name of the unit. If the schema is null, then the current user schema is used. If the status of the stored unit is `VALID` and the user has the required privilege, then the post-processed source text is guaranteed to be the same as that of the unit the last time it was compiled.

Subprograms of the second or third style are used to generate post-processed source text in the current user schema. The source text is passed in as a single `VARCHAR2` string in the second style, or as a `VARCHAR2` associative array in the third style. The source text can represent an arbitrary PL/SQL compilation unit. A typical usage is to pass the source text of an anonymous block and generate its post-processed source text in the current user schema. The third style can be useful when the source text exceeds the `VARCHAR2` length limit.

Operating notes

- For subprograms of the first style, the status of the stored PL/SQL unit is not required to be `VALID`. Likewise, the source text passed in as a `VARCHAR2` string or a `VARCHAR2` associative array may contain compile time errors. If errors are found when generating the post-processed source, the error message text will also appear at the end of the post-processed source text. In some cases, the preprocessing can be aborted because of errors. When this happens, the post-processed source text will appear to be incomplete and the associated error message can help indicate that an error has occurred during preprocessing.
- For subprograms of the second or third style, the source text can represent any arbitrary PL/SQL compilation unit. However, the source text of a valid PL/SQL compilation unit cannot include commonly used prefixes such as `CREATE OR REPLACE`. In general, the input source should be syntactically prepared in a way as if it were obtained from the `ALL_SOURCE` view. The following list gives some examples of valid initial syntax for some PL/SQL compilation units.

  ```
  anonymous block   (BEGIN | DECLARE) ...
  package           PACKAGE name ...
  package body      PACKAGE BODY name ...
  procedure         PROCEDURE name ...
  function          FUNCTION name ...
  ```

  If the source text represents a named PL/SQL unit that is valid, that unit will not be created after its post-processed source text is generated.
- If the text of a wrapped PL/SQL unit is obtained from the `ALL_SOURCE` view, the keyword `WRAPPED` always immediately follows the name of the unit, as in this example:

  ```
  PROCEDURE "some proc" WRAPPED
  a000000
  b2
  ...
  ```

  If such source text is presented to a GET_POST_PROCESSED_SOURCE function or a PRINT_POST_PROCESSED_SOURCE procedure, the exception `DBMS_PREPROCESSOR.WRAPPED_INPUT` is raised.

Data structures

The `DBMS_PREPROCESSOR` package defines a table type.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

Table types

SOURCE_LINES_T table type

SOURCE_LINES_T table type

This table type stores lines of post-processed source text. It is used to hold PL/SQL source text both before and after it is processed. It is especially useful in cases in which the amount of text exceeds 32K.

Syntax

```
TYPE source_lines_t IS
    TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

Summary of DBMS_PREPROCESSOR subprograms

Table 4–1 DBMS_PREPROCESSOR package subprograms

Subprogram	Description
GET_POST_PROCESSED_SOURCE function	Returns the post-processed source text.
PRINT_POST_PROCESSED_SOURCE procedure	Prints post-processed source text.

GET_POST_PROCESSED_SOURCE function

This overloaded function returns the post-processed source text. The different functionality of each form of syntax is presented along with the definition.

Syntax

Returns post-processed source text of a stored PL/SQL unit:

```
DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
   object_type    IN VARCHAR2,
   schema_name    IN VARCHAR2,
   object_name    IN VARCHAR2)
  RETURN dbms_preprocessor.source_lines_t;
```

Returns post-processed source text of a compilation unit:

```
DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
   source        IN VARCHAR2)
  RETURN dbms_preprocessor.source_lines_t;
```

Returns post-processed source text of an associative array (index-by table) containing the source text of the compilation unit:

```
DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
   source        IN dbms_preprocessor.source_lines_t)
  RETURN dbms_preprocessor.source_lines_t;
```

Parameters

Table 4–2 GET_POST_PROCESSED_SOURCE function parameters

Parameter	Description
object_type	Must be one of `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, or `FUNCTION`. Case sensitive.
schema_name	The schema name. Case insensitive unless a quoted identifier is used. If `NULL`, use current schema.
object_name	The name of the object. Case insensitive unless a quoted identifier is used.
source	The source text of the compilation unit.
source_lines_t	Associative array containing the source text of the compilation unit. The source text is a concatenation of all the non-null associative array elements in ascending index order.

Return values

The function returns an associative array containing the lines of the post-processed source text starting from index 1.

Usage notes

- Newline characters are not removed.
- Each line in the post-processed source text is mapped to a row in the associative array.
- In the post-processed source, unselected text will have blank lines.

Exceptions

Table 4–3 GET_POST_PROCESSED_SOURCE function exceptions

Exception	Description
`ORA-24234`	Insufficient privileges or object does not exist.
`ORA-24235`	Bad value for object type. Must be one of `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, or `FUNCTION`.
`ORA-24236`	The source text is empty.
`ORA-00931`	Missing identifier. The `object_name` value cannot be `NULL`.
`ORA-06502`	Numeric or value error: ▪ Character string buffer too small. ▪ A line is too long (more than 32767 bytes).

PRINT_POST_PROCESSED_SOURCE procedure

This overloaded procedure calls `DBMS_OUTPUT.PUT_LINE` to let you view post-processed source text. The different functionality of each form of syntax is presented along with the definition.

Syntax

Prints post-processed source text of a stored PL/SQL unit:

```
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
   object_type    IN VARCHAR2,
   schema_name    IN VARCHAR2,
   object_name    IN VARCHAR2);
```

Prints post-processed source text of a compilation unit:

```
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
   source        IN VARCHAR2);
```

Prints post-processed source text of an associative array containing the source text of the compilation unit:

```
DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
   source        IN dbms_preprocessor.source_lines_t);
```

Parameters

Table 4–4 PRINT_POST_PROCESSED_SOURCE procedure parameters

Parameter	Description
object_type	Must be one of `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, or `FUNCTION`. Case sensitive.
schema_name	The schema name. Case insensitive unless a quoted identifier is used. If `NULL`, use current schema.
object_name	The name of the object. Case insensitive unless a quoted identifier is used.
source	The source text of the compilation unit.
source_lines_t	Associative array containing the source text of the compilation unit. The source text is a concatenation of all the non-null associative array elements in ascending index order.

Usage notes

The associative array may contain holes. Null elements are ignored when doing the concatenation.

Exceptions

Table 4–5 PRINT_POST_PROCESSED_SOURCE procedure exceptions

Exception	Description
`ORA-24234`	Insufficient privileges or object does not exist.
`ORA-24235`	Bad value for object type. Must be `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, or `FUNCTION`.

Table 4–5 (Cont.) PRINT_POST_PROCESSED_SOURCE procedure exceptions

Exception	Description
`ORA-24236`	The source text is empty.
`ORA-00931`	Missing identifier. The `object_name` value cannot be `NULL`.
`ORA-06502`	Numeric or value error: ■ Character string buffer too small. ■ A line is too long (more than 32767 bytes).

5

DBMS_RANDOM

The `DBMS_RANDOM` package provides a built-in random number generator.

This chapter contains the following topics:

- Using DBMS_RANDOM
  - Operational notes
- Summary of DBMS_RANDOM subprograms

Note: `DBMS_RANDOM` is not intended for cryptography.

Using DBMS_RANDOM

- Operational notes

Operational notes

- `DBMS_RANDOM.RANDOM` produces integers in the range [-2^^31, 2^^31).
- `DBMS_RANDOM.VALUE` produces numbers in the range [0,1) with 38 digits of precision.

`DBMS_RANDOM` can be explicitly initialized but does not require initialization before a call to the random number generator. It will automatically initialize with the date, user ID, and process ID if no explicit initialization is performed.

If this package is seeded twice with the same seed, then accessed in the same way, it will produce the same result in both cases.

In some cases, such as when testing, you may want the sequence of random numbers to be the same on every run. In that case, you seed the generator with a constant value by calling an overload of `DBMS_RANDOM.SEED`. To produce different output for every run, simply omit the seed call and the system will choose a suitable seed for you.

Summary of DBMS_RANDOM subprograms

Table 5–1 DBMS_RANDOM package subprograms

Subprogram	Description
INITIALIZE procedure	Initializes the package with a seed value.
NORMAL function	Returns random numbers in a normal distribution.
RANDOM function	Generates a random number.
SEED procedure	Resets the seed.
STRING function	Gets a random string.
TERMINATE procedure	Terminates package.
VALUE function	One version gets a random number greater than or equal to 0 and less than 1, with 38 digits to the right of the decimal point (38-digit precision). The other version gets a random Oracle number `x`, where `x` is greater than or equal to a specified lower limit and less than a specified higher limit.

Note: The INITIALIZE procedure, RANDOM function and the TERMINATE procedure are deprecated. They are included in this release for legacy reasons only.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

INITIALIZE procedure

This procedure is deprecated. Although currently supported, it should not be used. It initializes the random number generator.

Syntax

```
DBMS_RANDOM.INITIALIZE (
   val  IN  BINARY_INTEGER);
```

Parameters

Table 5–2 INITIALIZE procedure parameters

Parameter	**Description**
val	The seed number used to generate a random number.

Usage notes

This procedure is obsolete as it simply calls the SEED procedure.

NORMAL function

This function returns random numbers in a standard normal distribution.

Syntax

```
DBMS_RANDOM.NORMAL
  RETURN NUMBER;
```

Return value

The random number, a `NUMBER` value.

RANDOM function

This procedure is deprecated. Although currently supported, it should not be used. It generates and returns a random number.

Syntax

```
DBMS_RANDOM.RANDOM
   RETURN binary_integer;
```

Return value

A random `BINARY_INTEGER` value greater than or equal to `-power(2,31)` and less than `power(2,31)`.

Usage notes

See the NORMAL function and the VALUE function.

SEED procedure

This procedure resets the seed used in generating a random number.

Syntax

```
DBMS_RANDOM.SEED (
   val  IN  BINARY_INTEGER);

DBMS_RANDOM.SEED (
   val  IN  VARCHAR2);
```

Parameters

Table 5–3 SEED procedure parameters

Parameter	Description
`val`	Seed number or string used to generate a random number.

Usage notes

The seed can be a string up to length 2000.

STRING function

This function generates and returns a random string.

Syntax

```
DBMS_RANDOM.STRING
   opt  IN  CHAR,
   len  IN  NUMBER)
  RETURN VARCHAR2;
```

Parameters

Table 5–4 STRING function parameters

Parameter	Description
opt	Specifies what the returning string looks like: - 'u', 'U' - returning string in uppercase alpha characters. - 'l', 'L' - returning string in lowercase alpha characters. - 'a', 'A' - returning string in mixed case alpha characters. - 'x', 'X' - returning string in uppercase alpha-numeric characters. - 'p', 'P' - returning string in any printable characters. Otherwise the returning string is in uppercase alpha characters.
len	The length of the returned string.

Return value

A `VARCHAR2` value with the random string.

TERMINATE procedure

This procedure is deprecated. Although currently supported, it should not be used. It would be called when the user is finished with the package.

Syntax

```
DBMS_RANDOM.TERMINATE;
```

VALUE function

One version returns a random number, greater than or equal to 0 and less than 1, with 38 digits to the right of the decimal (38-digit precision). The other version returns a random Oracle `NUMBER` value `x`, where `x` is greater than or equal to the specified *`low`* value and less than the specified *`high`* value.

Syntax

```
DBMS_RANDOM.VALUE
  RETURN NUMBER;

DBMS_RANDOM.VALUE(
  low  IN  NUMBER,
  high IN  NUMBER)
RETURN NUMBER;
```

Parameters

Table 5–5 VALUE function parameters

Parameter	Description
`low`	Lower limit of the range in which to generate a random number. The number generated is greater than or equal to *`low`*.
`high`	Higher limit of the range in which to generate a random number. The number generated is less than *`high`*.

Return value

A `NUMBER` value that is the generated random number.

6

DBMS_SQL

The `DBMS_SQL` package provides an interface to use dynamic SQL to parse any data manipulation language (DML) or data definition language (DDL) statement using PL/SQL. For example, you can enter a `DROP TABLE` statement from within a stored procedure by using the `PARSE` procedure supplied with the `DBMS_SQL` package.

This chapter contains the following topics:

- Using DBMS_SQL
  - Overview
  - Security model
  - Constants
  - Operational notes
  - Exceptions
  - Examples
- Data structures
  - Record types
  - Table types
- Summary of DBMS_SQL subprograms

Note: For more information on native dynamic SQL, see "Dynamic SQL in PL/SQL (EXECUTE IMMEDIATE statement)" in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* or see *Oracle Database PL/SQL Language Reference.*

Using DBMS_SQL

- Overview
- Security model
- Constants
- Exceptions
- Operational notes
- Examples

Overview

TimesTen lets you write stored procedures and anonymous PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are not embedded in your source program; rather, they are stored in character strings that are input to, or built by, the program at runtime. This functionality enables you to create more general-purpose procedures. For example, dynamic SQL lets you create a procedure that operates on a table whose name is not known until runtime.

Native dynamic SQL is an alternative to `DBMS_SQL` that lets you place dynamic SQL statements directly into PL/SQL blocks. In most situations, native dynamic SQL is easier to use and performs better than `DBMS_SQL`. However, native dynamic SQL itself has certain limitations, such as there being no support for so-called Method 4 (for dynamic SQL statements with an unknown number of inputs or outputs). Also, there are some tasks that can only be performed using `DBMS_SQL`.

The ability to use dynamic SQL from within stored procedures generally follows the model of the Oracle Call Interface (OCI). See *Oracle Call Interface Programmer's Guide* for information about OCI.

PL/SQL differs somewhat from other common programming languages, such as C. For example, addresses (also called *pointers*) are not user-visible in PL/SQL. As a result, there are some differences between OCI and the `DBMS_SQL` package, including the following:

- OCI binds by address, while the `DBMS_SQL` package binds by value.
- With `DBMS_SQL` you must call `VARIABLE_VALUE` to retrieve the value of an `OUT` parameter for an anonymous block, and you must call `COLUMN_VALUE` after fetching rows to actually retrieve the values of the columns in the rows into your program.
- The current release of the `DBMS_SQL` package does not provide `CANCEL` cursor procedures.
- Indicator variables are not required, because `NULL` is fully supported as a value of a PL/SQL variable.

Security model

`DBMS_SQL` is owned by `SYS` and compiled with `AUTHID CURRENT_USER`. Any `DBMS_SQL` subprogram called from an anonymous PL/SQL block is run using the privileges of the current user.

See "Definer's rights and invoker's rights" in *Oracle TimesTen In-Memory Database PL/SQL Developer's Guide* for information about the `AUTHID` clause.

Constants

The constants described in Table 6–1 are used with the `language_flag` parameter of the PARSE procedure. For TimesTen, use `NATIVE`.

Table 6–1 DBMS_SQL constants

Name	Type	Value	Description
`V6`	`INTEGER`	0	Specifies Oracle database version 6 behavior (not applicable for TimesTen).
`NATIVE`	`INTEGER`	1	Specifies normal behavior for the database to which the program is connected.
`V7`	`INTEGER`	2	Specifies Oracle database version 7 behavior (not applicable for TimesTen).

Operational notes

- Execution flow
- Processing queries
- Processing updates, inserts, and deletes
- Locating errors

Execution flow

1. OPEN_CURSOR
2. PARSE
3. BIND_VARIABLE or BIND_ARRAY
4. DEFINE_COLUMN or DEFINE_ARRAY
5. EXECUTE
6. FETCH_ROWS or EXECUTE_AND_FETCH
7. VARIABLE_VALUE or COLUMN_VALUE
8. CLOSE_CURSOR

OPEN_CURSOR

To process a SQL statement, you must have an open cursor. When you call the OPEN_CURSOR function, you receive a cursor ID number for the data structure representing a valid cursor maintained by TimesTen. These cursors are distinct from cursors defined at the precompiler, OCI, or PL/SQL level, and are used only by the `DBMS_SQL` package.

PARSE

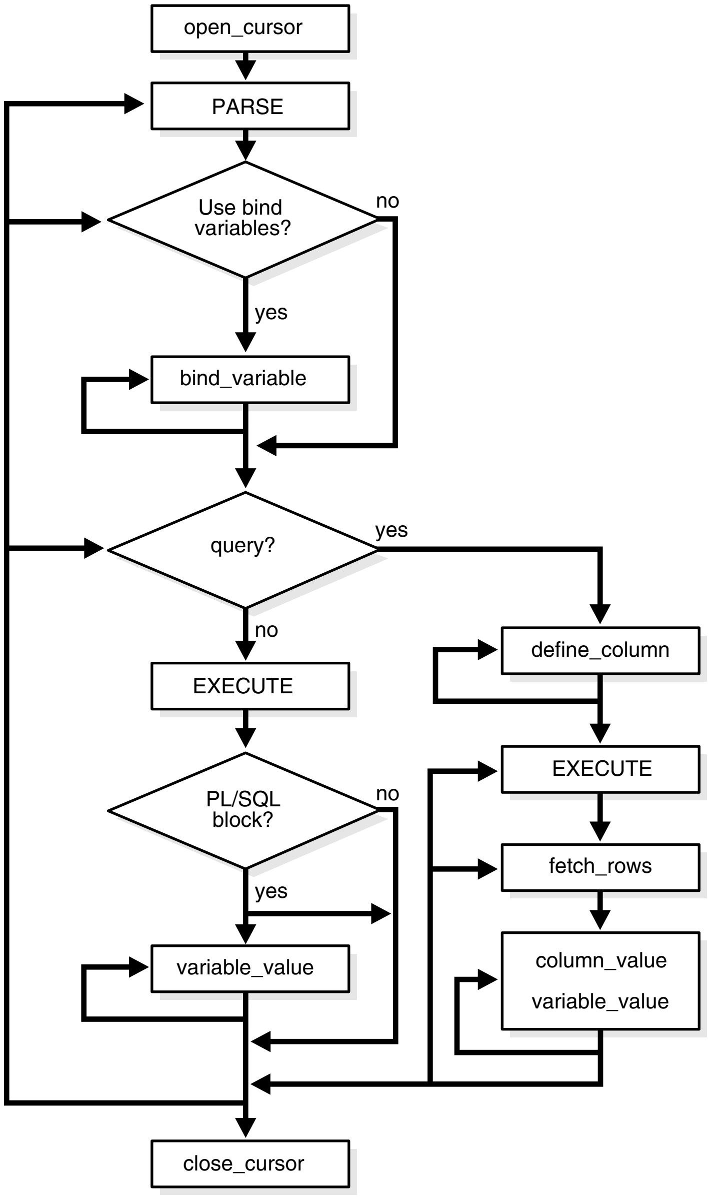
Every SQL statement must be parsed by calling the PARSE procedure. Parsing the statement checks the statement syntax and associates it with the cursor in your program.

You can parse any DML or DDL statement. DDL statements are run on the parse, which performs the implied commit.

> **Note:** When parsing a DDL statement to drop a procedure or a package, a timeout can occur if you are still using the procedure in question or a procedure in the package in question. After a call to a procedure, that procedure is considered to be in use until execution has returned to the user side. Any such timeout times out after a short time.

The execution flow of `DBMS_SQL` is shown in Figure 6–1 that follows.

Figure 6–1 DBMS_SQL execution flow



BIND_VARIABLE or BIND_ARRAY

Many DML statements require that data in your program be input to TimesTen. When you define a SQL statement that contains input data to be supplied at runtime, you must use placeholders in the SQL statement to mark where data must be supplied.

For each placeholder in the SQL statement, you must call a bind procedure, either the BIND_ARRAY procedure on page 6-43 or the BIND_VARIABLE procedure on page 6-45, to supply the value of a variable in your program (or the values of an array) to the placeholder. When the SQL statement is subsequently run, TimesTen uses the data that your program has placed in the output and input, or bind, variables.

`DBMS_SQL` can run a DML statement multiple times, each time with a different bind variable. The `BIND_ARRAY` procedure lets you bind a collection of scalars, each value of which is used as an input variable once for each `EXECUTE`. This is similar to the array interface supported by the OCI.

DEFINE_COLUMN or DEFINE_ARRAY

The columns of the row being selected in a `SELECT` statement are identified by their relative positions as they appear in the select list, from left to right. For a query, you must call a define procedure (`DEFINE_COLUMN` or `DEFINE_ARRAY`) to specify the variables that are to receive the `SELECT` values, much the way an `INTO` clause does for a static query.

Use the `DEFINE_ARRAY` procedure to define a PL/SQL collection into which rows will be fetched in a single `SELECT` statement. `DEFINE_ARRAY` provides an interface to fetch multiple rows at one fetch. You must call `DEFINE_ARRAY` before using the `COLUMN_VALUE` procedure to fetch the rows.

EXECUTE

Call the `EXECUTE` function to run your SQL statement.

FETCH_ROWS or EXECUTE_AND_FETCH

The `FETCH_ROWS` function retrieves the rows that satisfy the query. Each successive fetch retrieves another set of rows, until the fetch cannot retrieve any more rows. Instead of calling `EXECUTE` and then `FETCH_ROWS`, you may find it more efficient to call `EXECUTE_AND_FETCH` if you are calling `EXECUTE` for a single execution.

VARIABLE_VALUE or COLUMN_VALUE

For queries, call `COLUMN_VALUE` to determine the value of a column retrieved by the `FETCH_ROWS` call. For anonymous blocks containing calls to PL/SQL procedures or DML statements with a `RETURNING` clause, call `VARIABLE_VALUE` to retrieve the values assigned to the output variables when statements were run.

CLOSE_CURSOR

When you no longer need a cursor for a session, close the cursor by calling `CLOSE_CURSOR`.

If you neglect to close a cursor, then the memory used by that cursor remains allocated even though it is no longer needed.

Processing queries

If you are using dynamic SQL to process a query, then you must perform the following steps:

1. Specify the variables that are to receive the values returned by the `SELECT` statement by calling the DEFINE_COLUMN procedure or the DEFINE_ARRAY procedure.
2. Run your `SELECT` statement by calling the EXECUTE function.
3. Call the FETCH_ROWS function (or `EXECUTE_AND_FETCH`) to retrieve the rows that satisfied your query.
4. Call COLUMN_VALUE procedure to determine the value of a column retrieved by `FETCH_ROWS` for your query. If you used anonymous blocks containing calls to PL/SQL procedures, then you must call the VARIABLE_VALUE procedure to retrieve the values assigned to the output variables of these procedures.

Processing updates, inserts, and deletes

If you are using dynamic SQL to process an `INSERT`, `UPDATE`, or `DELETE`, then you must perform the following steps.

1. You must first run your `INSERT`, `UPDATE`, or `DELETE` statement by calling the EXECUTE function.

2. If statements have the `RETURNING` clause, then you must call the VARIABLE_VALUE procedure to retrieve the values assigned to the output variables.

Locating errors

There are additional functions in the `DBMS_SQL` package for obtaining information about the last referenced cursor in the session. The values returned by these functions are only meaningful immediately after a SQL statement is run. In addition, some error-locating functions are only meaningful after certain `DBMS_SQL` calls. For example, call the LAST_ERROR_POSITION function immediately after a `PARSE` call.

Exceptions

```
inconsistent_type EXCEPTION;
  pragma exception_init(inconsistent_type, -6562);
```

This exception is raised by the COLUMN_VALUE procedure or the VARIABLE_VALUE procedure when the type of the given `OUT` parameter (for where to put the requested value) is different from the type of the value.

Examples

This section provides example procedures that make use of the `DBMS_SQL` package.

Example 1

This example does not require the use of dynamic SQL because the text of the statement is known at compile time, but it illustrates the basic concept underlying the package.

The `demo` procedure deletes all employees from a table `myemployees` (created from the `employees` table of the `HR` schema) whose salaries exceed a specified value.

```
CREATE OR REPLACE PROCEDURE demo(p_salary IN NUMBER) AS
   cursor_name INTEGER;
   rows_processed INTEGER;
BEGIN
   cursor_name := dbms_sql.open_cursor;
   DBMS_SQL.PARSE(cursor_name, 'DELETE FROM myemployees WHERE salary > :x',
                  DBMS_SQL.NATIVE);
   DBMS_SQL.BIND_VARIABLE(cursor_name, ':x', p_salary);
   rows_processed := DBMS_SQL.EXECUTE(cursor_name);
   DBMS_SQL.CLOSE_CURSOR(cursor_name);
EXCEPTION
WHEN OTHERS THEN
   DBMS_SQL.CLOSE_CURSOR(cursor_name);
END;
/
```

Create the `myemployees` table and see how many employees have salaries greater than or equal to $15,000:

```
Command> create table myemployees as select * from employees;
107 rows inserted.

Command> select * from myemployees where salary>=15000;
< 100, Steven, King, SKING, 515.123.4567, 1987-06-17 00:00:00, AD_PRES, 24000,
<NULL>, <NULL>, 90 >
< 101, Neena, Kochhar, NKOCHHAR, 515.123.4568, 1989-09-21 00:00:00, AD_VP, 17000,
<NULL>, 100, 90 >
< 102, Lex, De Haan, LDEHAAN, 515.123.4569, 1993-01-13 00:00:00, AD_VP, 17000,
<NULL>, 100, 90 >
3 rows found.
```

Run `demo` to delete everyone with a salary greater than $14,999 and confirm the results:

```
Command> begin
       > demo(14999);
       > end;
       > /

PL/SQL procedure successfully completed.

Command> select * from myemployees where salary>=15000;
0 rows found.
```

Example 2

The following sample procedure is passed the names of a source and a destination table, and copies the rows from the source table to the destination table. This sample procedure assumes that both the source and destination tables have the following columns:

```
id        of type NUMBER
name      of type VARCHAR2(30)
birthdate of type DATE
```

This procedure does not specifically require the use of dynamic SQL; however, it illustrates the concepts of this package.

```
CREATE OR REPLACE PROCEDURE copy (
     source      IN VARCHAR2,
     destination IN VARCHAR2) IS
     id_var             NUMBER;
     name_var           VARCHAR2(30);
     birthdate_var      DATE;
     source_cursor      INTEGER;
     destination_cursor INTEGER;
     ignore             INTEGER;
  BEGIN

  -- Prepare a cursor to select from the source table:
     source_cursor := dbms_sql.open_cursor;
     DBMS_SQL.PARSE(source_cursor,
         'SELECT id, name, birthdate FROM ' || source,
          DBMS_SQL.NATIVE);
     DBMS_SQL.DEFINE_COLUMN(source_cursor, 1, id_var);
     DBMS_SQL.DEFINE_COLUMN(source_cursor, 2, name_var, 30);
     DBMS_SQL.DEFINE_COLUMN(source_cursor, 3, birthdate_var);
     ignore := DBMS_SQL.EXECUTE(source_cursor);

  -- Prepare a cursor to insert into the destination table:
     destination_cursor := DBMS_SQL.OPEN_CURSOR;
     DBMS_SQL.PARSE(destination_cursor,
                  'INSERT INTO ' || destination ||
                  ' VALUES (:id_bind, :name_bind, :birthdate_bind)',
                   DBMS_SQL.NATIVE);

  -- Fetch a row from the source table and insert it into the destination table:
     LOOP
       IF DBMS_SQL.FETCH_ROWS(source_cursor)>0 THEN
         -- get column values of the row
         DBMS_SQL.COLUMN_VALUE(source_cursor, 1, id_var);
         DBMS_SQL.COLUMN_VALUE(source_cursor, 2, name_var);
         DBMS_SQL.COLUMN_VALUE(source_cursor, 3, birthdate_var);

  -- Bind the row into the cursor that inserts into the destination table. You
  -- could alter this example to require the use of dynamic SQL by inserting an
  -- if condition before the bind.
        DBMS_SQL.BIND_VARIABLE(destination_cursor, ':id_bind', id_var);
        DBMS_SQL.BIND_VARIABLE(destination_cursor, ':name_bind', name_var);
        DBMS_SQL.BIND_VARIABLE(destination_cursor, ':birthdate_bind',
                               birthdate_var);
        ignore := DBMS_SQL.EXECUTE(destination_cursor);
      ELSE

  -- No more rows to copy:
        EXIT;
      END IF;
    END LOOP;
  -- Commit (in TimesTen commit closes cursors automatically):
     COMMIT;

    EXCEPTION
     WHEN OTHERS THEN
       IF DBMS_SQL.IS_OPEN(source_cursor) THEN
         DBMS_SQL.CLOSE_CURSOR(source_cursor);
       END IF;
       IF DBMS_SQL.IS_OPEN(destination_cursor) THEN
         DBMS_SQL.CLOSE_CURSOR(destination_cursor);
       END IF;
       RAISE;
  END;
/
```

Examples 3, 4, and 5: Bulk DML

This series of examples shows how to use bulk array binds (table items) in the SQL DML statements `INSERT`, `UPDATE`, and `DELETE`.

Here is an example of a bulk `INSERT` statement that adds three new departments to the `departments` table in the `HR` schema:

```
DECLARE
  stmt VARCHAR2(200);
  departid_array     DBMS_SQL.NUMBER_TABLE;
  deptname_array     DBMS_SQL.VARCHAR2_TABLE;
  mgrid_array        DBMS_SQL.NUMBER_TABLE;
  locid_array        DBMS_SQL.NUMBER_TABLE;
  c                NUMBER;
  dummy            NUMBER;
BEGIN
  departid_array(1):= 280;
  departid_array(2):= 290;
  departid_array(3):= 300;

  deptname_array(1) := 'Community Outreach';
  deptname_array(2) := 'Product Management';
  deptname_array(3) := 'Acquisitions';

  mgrid_array(1) := 121;
  mgrid_array(2) := 120;
  mgrid_array(3) := 70;

  locid_array(1):= 1500;
  locid_array(2):= 1700;
  locid_array(3):= 2700;

  stmt := 'INSERT INTO departments VALUES(
     :departid_array, :deptname_array, :mgrid_array, :locid_array)';
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_ARRAY(c, ':departid_array', departid_array);
  DBMS_SQL.BIND_ARRAY(c, ':deptname_array', deptname_array);
  DBMS_SQL.BIND_ARRAY(c, ':mgrid_array', mgrid_array);
  DBMS_SQL.BIND_ARRAY(c, ':locid_array', locid_array);
  dummy := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.CLOSE_CURSOR(c);
  EXCEPTION WHEN OTHERS THEN
    IF DBMS_SQL.IS_OPEN(c) THEN
      DBMS_SQL.CLOSE_CURSOR(c);
    END IF;
    RAISE;
END;
/
```

Here is output from a `SELECT` statement, showing the new rows:

```
Command> select * from departments;
< 10, Administration, 200, 1700 >
...
< 280, Community Outreach, 121, 1500 >
< 290, Product Management, 120, 1700 >
< 300, Acquisitions, 70, 2700 >
30 rows found.
```

Here is an example of a bulk `UPDATE` statement that demonstrates updating salaries for four existing employees in the `employees` table in the `HR` schema:

```
DECLARE
  stmt VARCHAR2(200);
  empno_array     DBMS_SQL.NUMBER_TABLE;
  salary_array    DBMS_SQL.NUMBER_TABLE;
  c               NUMBER;
  dummy           NUMBER;

BEGIN
  empno_array(1):= 203;
  empno_array(2):= 204;
  empno_array(3):= 205;
  empno_array(4):= 206;

  salary_array(1) := 7000;
  salary_array(2) := 11000;
  salary_array(3) := 13000;
  salary_array(4) := 9000;

  stmt := 'update employees set salary = :salary_array
    WHERE employee_id = :num_array';
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_ARRAY(c, ':num_array', empno_array);
  DBMS_SQL.BIND_ARRAY(c, ':salary_array', salary_array);
  dummy := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.CLOSE_CURSOR(c);

  EXCEPTION WHEN OTHERS THEN
    IF DBMS_SQL.IS_OPEN(c) THEN
      DBMS_SQL.CLOSE_CURSOR(c);
    END IF;
    RAISE;
END;
/
```

Assume the following entries for the specified employees before running the example, showing salaries of $6500, $10000, $12000, and $8300:

```
Command> select * from employees where employee_id>=203 and employee_id<=206;
< 203, Susan, Mavris, SMAVRIS, 515.123.7777, 1994-06-07 00:00:00, HR_REP,
6500, <NULL>, 101, 40 >
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00, PR_REP,
10000, <NULL>, 101, 70 >
```

```
< 205, Shelley, Higgins, SHIGGINS, 515.123.8080, 1994-06-07 00:00:00, AC_MGR,
12000, <NULL>, 101, 110 >
< 206, William, Gietz, WGIETZ, 515.123.8181, 1994-06-07 00:00:00, AC_ACCOUNT,
8300, <NULL>, 205, 110 >
4 rows found.
```

The following shows the new salaries after running the example:

```
Command> select * from employees where employee_id>=203 and employee_id<=206;
< 203, Susan, Mavris, SMAVRIS, 515.123.7777, 1994-06-07 00:00:00, HR_REP,
7000, <NULL>, 101, 40 >
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00, PR_REP,
11000, <NULL>, 101, 70 >
< 205, Shelley, Higgins, SHIGGINS, 515.123.8080, 1994-06-07 00:00:00, AC_MGR,
13000, <NULL>, 101, 110 >
< 206, William, Gietz, WGIETZ, 515.123.8181, 1994-06-07 00:00:00, AC_ACCOUNT,
9000, <NULL>, 205, 110 >
4 rows found.
```

In a `DELETE` statement, for example, you could bind in an array in the `WHERE` clause and have the statement be run for each element in the array, as follows:

```
DECLARE
  stmt VARCHAR2(200);
  dept_no_array DBMS_SQL.NUMBER_TABLE;
  c NUMBER;
  dummy NUMBER;
BEGIN
  dept_no_array(1) := 60;
  dept_no_array(2) := 70;
  stmt := 'delete from employees where department_id = :dept_array';
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_ARRAY(c, ':dept_array', dept_no_array, 1, 1);
  dummy := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.CLOSE_CURSOR(c);

  EXCEPTION WHEN OTHERS THEN
    IF DBMS_SQL.IS_OPEN(c) THEN
      DBMS_SQL.CLOSE_CURSOR(c);
    END IF;
    RAISE;
END;
/
```

In this example, only the first element of the array is specified by the `BIND_ARRAY` call (lower and upper bounds of the array elements are both set to 1), so only employees in department 60 are deleted.

Before running the example, there are five employees in department 60 and one in department 70, where the department number is the last entry in each row:

```
Command> select * from employees where department_id>=60 and department_id<=70;
< 103, Alexander, Hunold, AHUNOLD, 590.423.4567, 1990-01-03 00:00:00, IT_PROG,
9000, <NULL>, 102, 60 >
< 104, Bruce, Ernst, BERNST, 590.423.4568, 1991-05-21 00:00:00, IT_PROG, 6000,
<NULL>, 103, 60 >
< 105, David, Austin, DAUSTIN, 590.423.4569, 1997-06-25 00:00:00, IT_PROG, 4800,
 <NULL>, 103, 60 >
< 106, Valli, Pataballa, VPATABAL, 590.423.4560, 1998-02-05 00:00:00, IT_PROG,
4800, <NULL>, 103, 60 >
```

```
< 107, Diana, Lorentz, DLORENTZ, 590.423.5567, 1999-02-07 00:00:00, IT_PROG,
4200, <NULL>, 103, 60 >
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00, PR_REP, 10000,
<NULL>, 101, 70 >
6 rows found.
```

After running the example, only the employee in department 70 remains:

```
Command> select * from employees where department_id>=60 and department_id<=70;
< 204, Hermann, Baer, HBAER, 515.123.8888, 1994-06-07 00:00:00, PR_REP, 10000,
<NULL>, 101, 70 >
1 row found.
```

Example 6: Defining an array

```
CREATE OR REPLACE PROCEDURE BULK_PLSQL(deptid NUMBER) IS
    names    DBMS_SQL.VARCHAR2_TABLE;
    sals     DBMS_SQL.NUMBER_TABLE;
    c        NUMBER;
    r        NUMBER;
    sql_stmt VARCHAR2(32767) :=
        'SELECT last_name, salary FROM employees WHERE department_id = :b1';

BEGIN
    c := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE(c, sql_stmt, dbms_sql.native);
    DBMS_SQL.BIND_VARIABLE(c, 'b1', deptid);
    DBMS_SQL.DEFINE_ARRAY(c, 1, names, 5, 1);
    DBMS_SQL.DEFINE_ARRAY(c, 2, sals, 5, 1);

    r := DBMS_SQL.EXECUTE(c);

    LOOP
      r := DBMS_SQL.FETCH_ROWS(c);
      DBMS_SQL.COLUMN_VALUE(c, 1, names);
      DBMS_SQL.COLUMN_VALUE(c, 2, sals);
      EXIT WHEN r != 5;
    END LOOP;

    DBMS_SQL.CLOSE_CURSOR(c);

    -- loop through the names and sals collections
    FOR i IN names.FIRST .. names.LAST  LOOP
      DBMS_OUTPUT.PUT_LINE('Name = ' || names(i) || ', salary = ' || sals(i));
    END LOOP;
END;
/
```

For example, for department 20 in the `employees` table, this produces the following output:

```
Command> begin
       > bulk_plsql(20);
       > end;
       > /
Name = Hartstein, salary = 13000
Name = Fay, salary = 6000

PL/SQL procedure successfully completed.
```

Example 7: Describe columns

This can be used as a substitute for the `ttIsql` `DESCRIBE` command by using a `SELECT *` query on the table to describe. This example describes columns of the `employees` table.

```
DECLARE
  c           NUMBER;
  d           NUMBER;
  col_cnt     INTEGER;
  f           BOOLEAN;
  rec_tab     DBMS_SQL.DESC_TAB;
  col_num    NUMBER;
  PROCEDURE print_rec(rec in DBMS_SQL.DESC_REC) IS
  BEGIN
    DBMS_OUTPUT.NEW_LINE;
    DBMS_OUTPUT.PUT_LINE('col_type            =    '
                         || rec.col_type);
    DBMS_OUTPUT.PUT_LINE('col_maxlen          =    '
                         || rec.col_max_len);
    DBMS_OUTPUT.PUT_LINE('col_name            =    '
                         || rec.col_name);
    DBMS_OUTPUT.PUT_LINE('col_name_len        =    '
                         || rec.col_name_len);
    DBMS_OUTPUT.PUT_LINE('col_schema_name     =    '
                         || rec.col_schema_name);
    DBMS_OUTPUT.PUT_LINE('col_schema_name_len =    '
                         || rec.col_schema_name_len);
    DBMS_OUTPUT.PUT_LINE('col_precision       =    '
                         || rec.col_precision);
    DBMS_OUTPUT.PUT_LINE('col_scale           =    '
                         || rec.col_scale);
    DBMS_OUTPUT.PUT('col_null_ok         =    ');
    IF (rec.col_null_ok) THEN
      DBMS_OUTPUT.PUT_LINE('true');
    ELSE
      DBMS_OUTPUT.PUT_LINE('false');
    END IF;
  END;
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;

  DBMS_SQL.PARSE(c, 'SELECT * FROM employees', DBMS_SQL.NATIVE);

  d := DBMS_SQL.EXECUTE(c);

  DBMS_SQL.DESCRIBE_COLUMNS(c, col_cnt, rec_tab);

/*
 * Following loop could simply be for j in 1..col_cnt loop.
 * Here we are simply illustrating some PL/SQL table
 * features.
 */
  col_num := rec_tab.first;
  IF (col_num IS NOT NULL) THEN
    LOOP
      print_rec(rec_tab(col_num));
      col_num := rec_tab.next(col_num);
      EXIT WHEN (col_num IS NULL);
    END LOOP;
  END IF;
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/
```

Here is an abbreviated sample of the output, describing columns of the `employees` table, assuming it was run from the `HR` schema. Information from only the first two columns is shown here:

```
col_type            =    2
col_maxlen          =    7
col_name            =    EMPLOYEE_ID
col_name_len        =    11
col_schema_name     =    HR
col_schema_name_len =    8
col_precision       =    6
col_scale           =    0
col_null_ok         =    false
col_type            =    1
col_maxlen          =    20
col_name            =    FIRST_NAME
col_name_len        =    10
col_schema_name     =    HR
col_schema_name_len =    8
col_precision       =    0
col_scale           =    0
col_null_ok         =    true
...
```

Example 8: RETURNING clause

With this clause, `INSERT`, `UPDATE`, and `DELETE` statements can return values of expressions. These values are returned in bind variables.

`DBMS_SQL.BIND_VARIABLE` is used to bind these outbinds if a single row is inserted, updated, or deleted. If multiple rows are inserted, updated, or deleted, then `DBMS_SQL.BIND_ARRAY` is used. `DBMS_SQL.VARIABLE_VALUE` must be called to get the values in these bind variables.

Note: This is similar to `DBMS_SQL.VARIABLE_VALUE`, which must be called after running a PL/SQL block with an out-bind inside `DBMS_SQL`.

The examples that follow assume a table `tab` has been created:

```
Command> create table tab (c1 number, c2 number);
```

i) Single row insert.

```
      CREATE OR REPLACE PROCEDURE single_Row_insert
           (c1 NUMBER, c2 NUMBER, r OUT NUMBER) is
      c NUMBER;
      n NUMBER;
      BEGIN
        c := DBMS_SQL.OPEN_CURSOR;
        DBMS_SQL.PARSE(c, 'INSERT INTO tab VALUES (:bnd1, :bnd2) ' ||
                          'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
     DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1);
        DBMS_SQL.BIND_VARIABLE(c, 'bnd2', c2);
        DBMS_SQL.BIND_VARIABLE(c, 'bnd3', r);
```

```
  n := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r); -- get value of outbind variable
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/
```

The following runs this example and shows the results. The table was initially empty.

```
Command> declare r NUMBER;
       > begin
       > single_Row_insert(100,200,r);
       > dbms_output.put_line('Product = ' || r);
       > end;
       > /
Product = 20000

PL/SQL procedure successfully completed.

Command> select * from tab;
< 100, 200 >
1 row found.
```

ii) Single row update. Note that `rownum` is an internal variable for row number.

```
CREATE OR REPLACE PROCEDURE single_Row_update
     (c1 NUMBER, c2 NUMBER, r out NUMBER) IS
c NUMBER;
n NUMBER;

BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'UPDATE tab SET c1 = :bnd1, c2 = :bnd2 ' ||
                    'WHERE rownum = 1 ' ||
                    'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd2', c2);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd3', r);
  n := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r);-- get value of outbind variable
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/
```

The following runs this example and shows the results, updating the row that was inserted in the previous example.

```
Command> declare r NUMBER;
       > begin
       > single_Row_update(200,300,r);
       > dbms_output.put_line('Product = ' || r);
       > end;
       > /
Product = 60000

PL/SQL procedure successfully completed.

Command> select * from tab;
< 200, 300 >
1 row found.
```

iii) Multiple row insert.

```
CREATE OR REPLACE PROCEDURE multi_Row_insert
     (c1 DBMS_SQL.NUMBER_TABLE, c2 DBMS_SQL.NUMBER_TABLE,
      r OUT DBMS_SQL.NUMBER_TABLE) is
c NUMBER;
n NUMBER;
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'insert into tab VALUES (:bnd1, :bnd2) ' ||
                    'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_ARRAY(c, 'bnd1', c1);
  DBMS_SQL.BIND_ARRAY(c, 'bnd2', c2);
  DBMS_SQL.BIND_ARRAY(c, 'bnd3', r);
  n := DBMS_SQL.EXECUTE(c);
  DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r);-- get value of outbind variable
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/
```

The following script runs this example:

```
declare
   c1_array dbms_sql.number_table;
   c2_array dbms_sql.number_table;
   r_array dbms_sql.number_table;
begin
   c1_array(1) := 10;
   c1_array(2) := 20;
   c1_array(3) := 30;
   c2_array(1) := 15;
   c2_array(2) := 25;
   c2_array(3) := 35;
   multi_Row_insert(c1_array,c2_array,r_array);
   dbms_output.put_line('Product for row1 = ' || r_array(1));
   dbms_output.put_line('Product for row2 = ' || r_array(2));
   dbms_output.put_line('Product for row3 = ' || r_array(3));
end;
/
```

Following are the results. The table was initially empty.

```
Product for row1 = 150
Product for row2 = 500
Product for row3 = 1050

PL/SQL procedure successfully completed.

Command> select * from tab;
< 10, 15 >
< 20, 25 >
< 30, 35 >
3 rows found.
```

iv) Multiple row update.

```
 CREATE OR REPLACE PROCEDURE multi_Row_update
      (c1 NUMBER, c2 NUMBER, r OUT DBMS_SQL.NUMBER_TABLE) IS
   c NUMBER;
   n NUMBER;

BEGIN
   c := DBMS_SQL.OPEN_CURSOR;
```

```
      DBMS_SQL.PARSE(c, 'UPDATE tab SET c1 = :bnd1 WHERE c2 > :bnd2 ' ||
                        'RETURNING c1*c2 INTO :bnd3', DBMS_SQL.NATIVE);
      DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1);
      DBMS_SQL.BIND_VARIABLE(c, 'bnd2', c2);
      DBMS_SQL.BIND_ARRAY(c, 'bnd3', r);
      n := DBMS_SQL.EXECUTE(c);
      DBMS_OUTPUT.PUT_LINE(n || ' rows updated');
      DBMS_SQL.VARIABLE_VALUE(c, 'bnd3', r);-- get value of outbind variable
      DBMS_SQL.CLOSE_CURSOR(c);
    END;
    /
```

Note: Note that `bnd1` and `bnd2` can be arrays as well. The value of the expression for all the rows updated will be in `bnd3`. There is no way of differentiating which rows were updated of each value of `bnd1` and `bnd2`.

The following script runs the example:

```
declare
   c1 NUMBER;
   c2 NUMBER;
   r_array dbms_sql.number_table;
begin
   c1 := 100;
   c2 := 0;
   multi_Row_update(c1, c2, r_array);
   dbms_output.put_line('Product for row1 = ' || r_array(1));
   dbms_output.put_line('Product for row2 = ' || r_array(2));
   dbms_output.put_line('Product for row3 = ' || r_array(3));
end;
/
```

Here are the results, updating the rows that were inserted in the previous example. (The report of the number of rows updated is from the example itself. The products are reported by the test script.)

```
3 rows updated
Product for row1 = 1500
Product for row2 = 2500
Product for row3 = 3500

PL/SQL procedure successfully completed.

Command> select * from tab;
< 100, 15 >
< 100, 25 >
< 100, 35 >
3 rows found.
Command>
```

v) Multiple row delete.

```
    CREATE OR REPLACE PROCEDURE multi_Row_delete
         (c1_test NUMBER,
          r OUT DBMS_SQL.NUMBER_TABLE) is
    c NUMBER;
    n NUMBER;
```

```
BEGIN
  c := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(c, 'DELETE FROM tab WHERE c1 = :bnd1 ' ||
                  'RETURNING c1*c2 INTO :bnd2', DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(c, 'bnd1', c1_test);
  DBMS_SQL.BIND_ARRAY(c, 'bnd2', r);
  n := DBMS_SQL.EXECUTE(c);
  DBMS_OUTPUT.PUT_LINE(n || ' rows deleted');
  DBMS_SQL.VARIABLE_VALUE(c, 'bnd2', r);-- get value of outbind variable
  DBMS_SQL.CLOSE_CURSOR(c);
END;
/
```

The following script runs the example.

```
declare
   c1_test NUMBER;
   r_array dbms_sql.number_table;
begin
   c1_test := 100;
   multi_Row_delete(c1_test, r_array);
   dbms_output.put_line('Product for row1 = ' || r_array(1));
   dbms_output.put_line('Product for row2 = ' || r_array(2));
   dbms_output.put_line('Product for row3 = ' || r_array(3));
end;
/
```

Here are the results, deleting the rows that were updated in the previous example. (The report of the number of rows deleted is from the example itself. The products are reported by the test script.)

```
3 rows deleted
Product for row1 = 1500
Product for row2 = 2500
Product for row3 = 3500

PL/SQL procedure successfully completed.

Command> select * from tab;
0 rows found.
```

Note: `DBMS_SQL.BIND_ARRAY` of `Number_Table` internally binds a number. The number of times statement is run depends on the number of elements in an inbind array.

Data structures

The `DBMS_SQL` package defines the following record types and table types.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

Record types

- DESC_REC record type
- DESC_REC2 record type
- DESC_REC3 record type

Table types

- BINARY_DOUBLE_TABLE table type
- BINARY_FLOAT_TABLE table type
- DATE_TABLE table type
- DESC_TAB table type
- DESC_TAB2 table type
- DESC_TAB3 table type
- INTERVAL_DAY_TO_SECOND_TABLE table type
- INTERVAL_YEAR_TO_MONTH_TABLE table type
- NUMBER_TABLE table type
- TIME_TABLE table type
- TIMESTAMP_TABLE table type
- VARCHAR2_TABLE table type
- VARCHAR2A table type
- VARCHAR2S table type

DESC_REC record type

Note: This type has been deprecated in favor of the DESC_REC2 record type.

This record type holds the describe information for a single column in a dynamic query. It is the element type of the `DESC_TAB` table type and the DESCRIBE_COLUMNS procedure.

Syntax

```
TYPE desc_rec IS RECORD (
      col_type            BINARY_INTEGER := 0,
      col_max_len         BINARY_INTEGER := 0,
      col_name            VARCHAR2(32)   := '',
      col_name_len        BINARY_INTEGER := 0,
      col_schema_name     VARCHAR2(32)   := '',
      col_schema_name_len BINARY_INTEGER := 0,
      col_precision       BINARY_INTEGER := 0,
      col_scale           BINARY_INTEGER := 0,
      col_charsetid       BINARY_INTEGER := 0,
      col_charsetform     BINARY_INTEGER := 0,
      col_null_ok         BOOLEAN        := TRUE);
TYPE desc_tab IS TABLE OF desc_rec INDEX BY BINARY_INTEGER;
```

Fields

Table 6–2 DESC_REC fields

Field	Description
col_type	Type of column.
col_max_len	Maximum column length.
col_name	Name of column.
col_name_len	Length of column name.
col_schema_name	Column schema name.
col_schema_name_len	Length of column schema name.
col_precision	Precision of column.
col_scale	Scale of column.
col_charsetid	Column character set id.
col_charsetform	Column character set form.
col_null_ok	Null column flag. `TRUE` if `NULL` is allowable.

DESC_REC2 record type

`DESC_REC2` is the element type of the `DESC_TAB2` table type and the DESCRIBE_COLUMNS2 procedure.

This record type is identical to `DESC_REC` except for the *`col_name`* field, which has been expanded to the maximum possible size for `VARCHAR2`. It is therefore preferred to `DESC_REC`, which is deprecated, because column name values can be greater than 32 characters.

Syntax

```
TYPE desc_rec2 IS RECORD (
   col_type            binary_integer := 0,
   col_max_len         binary_integer := 0,
   col_name            varchar2(32767) := '',
   col_name_len        binary_integer := 0,
   col_schema_name     varchar2(32)   := '',
   col_schema_name_len binary_integer := 0,
   col_precision       binary_integer := 0,
   col_scale           binary_integer := 0,
   col_charsetid       binary_integer := 0,
   col_charsetform     binary_integer := 0,
   col_null_ok         boolean        := TRUE);
```

Fields

Table 6–3   DESC_REC2 fields

Field	Description
`col_type`	Type of column.
`col_max_len`	Maximum column length.
`col_name`	Name of column.
`col_name_len`	Length of column name.
`col_schema_name`	Column schema name.
`col_schema_name_len`	Length of column schema name.
`col_precision`	Precision of column.
`col_scale`	Scale of column.
`col_charsetid`	Column character set ID.
`col_charsetform`	Column character set form.
`col_null_ok`	Null column flag. `TRUE` if `NULL` is allowable.

DESC_REC3 record type

`DESC_REC3` is the element type of the `DESC_TAB3` table type and the DESCRIBE_COLUMNS3 procedure.

`DESC_REC3` is identical to `DESC_REC2` except for two additional fields to hold the type name (*`type_name`*) and type name len (*`type_name_len`*) of a column in a dynamic query. The *`col_type_name`* and *`col_type_name_len`* fields are only populated when the *`col_type`* field value is 109 (the Oracle Database type number for user-defined types), which is not currently used.

Syntax

```
TYPE desc_rec3 IS RECORD (
   col_type              binary_integer := 0,
   col_max_len           binary_integer := 0,
   col_name              varchar2(32767) := '',
   col_name_len          binary_integer := 0,
   col_schema_name       varchar2(32) := '',
   col_schema_name_len   binary_integer := 0,
   col_precision         binary_integer := 0,
   col_scale             binary_integer := 0,
   col_charsetid         binary_integer := 0,
   col_charsetform       binary_integer := 0,
   col_null_ok           boolean := TRUE,
   col_type_name         varchar2(32767)   := '',
   col_type_name_len     binary_integer := 0);
```

Fields

Table 6–4 DESC_REC3 fields

Field	Description
`col_type`	Type of column.
`col_max_len`	Maximum column length.
`col_name`	Name of column.
`col_name_len`	Length of column name.
`col_schema_name`	Column schema name.
`col_schema_name_len`	Length of column schema name.
`col_precision`	Precision of column.
`col_scale`	Scale of column.
`col_charsetid`	Column character set ID.
`col_charsetform`	Column character set form.
`col_null_ok`	Null column flag. `TRUE` if `NULL` is allowable.
`col_type_name`	Reserved for future use.
`col_type_name_len`	Reserved for future use.

BINARY_DOUBLE_TABLE table type

This is a table of `BINARY_DOUBLE`.

Syntax

```
TYPE binary_double_table IS TABLE OF BINARY_DOUBLE INDEX BY BINARY_INTEGER;
```

BINARY_FLOAT_TABLE table type

This is a table of `BINARY_FLOAT`.

Syntax

```
TYPE binary_float_table IS TABLE OF BINARY_FLOAT INDEX BY BINARY_INTEGER;
```

DATE_TABLE table type

This is a table of `DATE`.

Syntax

```
type date_table IS TABLE OF DATE INDEX BY BINARY_INTEGER;
```

DESC_TAB table type

This is a table of DESC_REC record type.

Syntax

```
TYPE desc_tab IS TABLE OF desc_rec INDEX BY BINARY_INTEGER;
```

DESC_TAB2 table type

This is a table of DESC_REC2 record type.

Syntax

```
TYPE desc_tab2 IS TABLE OF desc_rec2 INDEX BY BINARY_INTEGER;
```

DESC_TAB3 table type

This is a table of DESC_REC3 record type.

Syntax

```
TYPE desc_tab3 IS TABLE OF desc_rec3 INDEX BY BINARY_INTEGER;
```

INTERVAL_DAY_TO_SECOND_TABLE table type

This is a table of `DSINTERVAL_UNCONSTRAINED`.

Syntax

```
TYPE interval_day_to_second_Table IS TABLE OF
   DSINTERVAL_UNCONSTRAINED INDEX BY binary_integer;
```

INTERVAL_YEAR_TO_MONTH_TABLE table type

This is a table of `YMINTERVAL_UNCONSTRAINED`.

Syntax

```
TYPE interval_year_to_month_table IS TABLE OF YMINTERVAL_UNCONSTRAINED
   INDEX BY BINARY_INTEGER;
```

NUMBER_TABLE table type

This is a table of `NUMBER`.

Syntax

```
TYPE number_table IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```

TIME_TABLE table type

This is a table of `TIME_UNCONSTRAINED`.

Syntax

```
TYPE time_table IS TABLE OF TIME_UNCONSTRAINED INDEX BY BINARY_INTEGER;
```

TIMESTAMP_TABLE table type

This is a table of `TIMESTAMP_UNCONSTRAINED`.

Syntax

```
TYPE timestamp_table IS TABLE OF TIMESTAMP_UNCONSTRAINED INDEX BY BINARY_INTEGER;
```

VARCHAR2_TABLE table type

This is table of `VARCHAR2(2000)`.

Syntax

```
TYPE varchar2_table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

VARCHAR2A table type

This is table of `VARCHAR2(32767)`.

Syntax

```
TYPE varchar2a IS TABLE OF VARCHAR2(32767) INDEX BY BINARY_INTEGER;
```

VARCHAR2S table type

This is table of `VARCHAR2(256)`.

Note: This type has been superseded by the VARCHAR2A table type. It is supported only for backward compatibility.

Syntax

```
TYPE varchar2s IS TABLE OF VARCHAR2(256) INDEX BY BINARY_INTEGER;
```

Summary of DBMS_SQL subprograms

Table 6–5 DBMS_SQL Package Subprograms

Subprogram	Description
BIND_ARRAY procedure	Binds a given value to a given collection.
BIND_VARIABLE procedure	Binds a given value to a given variable.
CLOSE_CURSOR procedure	Closes given cursor and frees memory.
COLUMN_VALUE procedure	Returns value of the cursor element for a given position in a cursor.
COLUMN_VALUE_LONG Procedure	Returns a selected part of a `LONG` column that has been defined using `DEFINE_COLUMN_LONG`. **Important**: Because TimesTen does not support the `LONG` data type, attempting to use this procedure in TimesTen will result in an `ORA-01018` error at runtime. The `COLUMN_VALUE_LONG` and `DEFINE_COLUMN_LONG` procedures are therefore not documented in this manual.
DEFINE_ARRAY procedure	Defines a collection to be selected from the given cursor. Used only with `SELECT` statements.
DEFINE_COLUMN procedure	Defines a column to be selected from the given cursor. Used only with `SELECT` statements.
DEFINE_COLUMN_LONG Procedure	Defines a `LONG` column to be selected from the given cursor. Used with `SELECT` statements. **Important**: Because TimesTen does not support the `LONG` data type, attempting to use the `COLUMN_VALUE_LONG` procedure in TimesTen will result in an `ORA-01018` error at runtime. The `COLUMN_VALUE_LONG` and `DEFINE_COLUMN_LONG` procedures are therefore not documented in this manual.
DESCRIBE_COLUMNS procedure	Describes the columns for a cursor opened and parsed through `DBMS_SQL`.
DESCRIBE_COLUMNS2 procedure	Describes the specified column, as an alternative to DESCRIBE_COLUMNS procedure.
DESCRIBE_COLUMNS3 procedure	Describes the specified column, as an alternative to DESCRIBE_COLUMNS procedure.
EXECUTE function	Executes a given cursor.
EXECUTE_AND_FETCH function	Executes a given cursor and fetches rows.
FETCH_ROWS function	Fetches a row from a given cursor.
IS_OPEN function	Returns `TRUE` if the given cursor is open.
LAST_ERROR_POSITION function	Returns byte offset in the SQL statement text where the error occurred.
LAST_ROW_COUNT function	Returns cumulative count of the number of rows fetched.

Table 6–5 (Cont.) DBMS_SQL Package Subprograms

Subprogram	Description
LAST_ROW_ID function	Returns rowid of last row processed. TimesTen does not support rowid of the last row operated on by a DML statement. This function returns `NULL`.
LAST_SQL_FUNCTION_CODE function	Returns SQL function code for statement.
OPEN_CURSOR function	Returns cursor ID number of new cursor.
PARSE procedure	Parses given statement.
TO_CURSOR_NUMBER function	Takes an opened strongly or weakly-typed REF CURSOR and transforms it into a `DBMS_SQL` cursor number.
TO_REFCURSOR function	Takes an opened, parsed, and executed cursor and transforms or migrates it into a PL/SQL-manageable REF CURSOR (a weakly typed cursor) that can be consumed by PL/SQL native dynamic SQL.
VARIABLE_VALUE procedure	Returns value of named variable for given cursor.

BIND_ARRAY procedure

This procedure binds a given value or set of values to a given variable in a cursor, based on the name of the variable in the statement.

Syntax

```
DBMS_SQL.BIND_ARRAY (
   c                   IN INTEGER,
   name                IN VARCHAR2,
   <table_variable>    IN <datatype>
 [,index1              IN INTEGER,
   index2              IN INTEGER)] );
```

Where the `table_variable` and its corresponding `datatype` can be any of the following matching pairs:

```
<bflt_tab>     dbms_sql.Binary_Float_Table
<bdbl_tab>     dbms_sql.Binary_Double_Table
<c_tab>        dbms_sql.Varchar2_Table
<d_tab>        dbms_sql.Date_Table
<ids_tab>      dbms_sql.Interval_Day_to_Second_Table
<iym_tab>      dbms_sql.Interval_Year_to_Month_Table
<n_tab>        dbms_sql.Number_Table
<tm_tab>       dbms_sql.Time_Table
<tms_tab>      dbms_sql.Timestamp_Table
```

Notice that the `BIND_ARRAY` procedure is overloaded to accept different data types.

Parameters

Table 6–6 BIND_ARRAY procedure parameters

Parameter	Description
c	ID number of the cursor where the value is to be bound.
name	Name of the collection in the statement.
table_variable	Local variable that has been declared as *datatype*.
index1	Index for the table element that marks the lower bound of the range.
index2	Index for the table element that marks the upper bound of the range.

Usage notes

The length of the bind variable name should be less than or equal to 30 bytes.

For binding a range, the table must contain the elements that specify the range—`tab(`*index1*`)` and `tab(`*index2*`)`—but the range does not have to be dense. The *index1* value must be less than or equal to *index2*. All elements between `tab(`*index1*`)` and `tab(`*index2*`)` are used in the bind.

If you do not specify indexes in the bind call, and two different binds in a statement specify tables that contain a different number of elements, then the number of elements actually used is the minimum number between all tables. This is also the case if you specify indexes. The minimum range is selected between the two indexes for all tables.

Not all bind variables in a query have to be array binds. Some can be regular binds and the same value are used for each element of the collections in expression evaluations (and so forth).

Bulk array binds

Bulk selects, inserts, updates, and deletes can enhance the performance of applications by bundling many calls into one. The `DBMS_SQL` package lets you work on collections of data using the PL/SQL table type.

Table items are unbounded homogeneous collections. In persistent storage, they are like other relational tables and have no intrinsic ordering. But when a table item is brought into the workspace (either by querying or by navigational access of persistent data), or when it is created as the value of a PL/SQL variable or parameter, its elements are given subscripts that can be used with array-style syntax to get and set the values of elements.

The subscripts of these elements need not be dense, and can be any number including negative numbers. For example, a table item can contain elements at locations -10, 2, and 7 only.

When a table item is moved from transient workspace to persistent storage, the subscripts are not stored. The table item is unordered in persistent storage.

At bind time the table is copied out from the PL/SQL buffers into local `DBMS_SQL` buffers (the same as for all scalar types), then the table is manipulated from the local `DBMS_SQL` buffers. Therefore, if you change the table after the bind call, then that change does not affect the way the execute acts.

Types for scalar collections

You can declare a local variable as one of the following table-item types, which are defined as public types in `DBMS_SQL`.

```
TYPE binary_double_table
                    IS TABLE OF BINARY_DOUBLE  INDEX BY BINARY_INTEGER;
TYPE binary_float_table
                    IS TABLE OF BINARY_FLOAT   INDEX BY BINARY_INTEGER;
TYPE date_table     IS TABLE OF DATE           INDEX BY BINARY_INTEGER;
TYPE interval_day_to_second_table
                    IS TABLE OF dsinterval_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE interval_year_to_month_table
                    IS TABLE OF yminterval_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE number_table   IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;
TYPE time_table     IS TABLE OF time_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE timestamp_table
                    IS TABLE OF timestamp_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE varchar2_table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;

<tm_tab>   Time_Table
<tms_tab>  Timestamp_Table
<ids_tab>  Interval_Day_To_Second_Table
<iym_tab>  Interval_Year_To_Month_Table
```

Examples

See "Examples" on page 6-11.

BIND_VARIABLE procedure

This procedures binds a given value or set of values to a given variable in a cursor, based on the name of the variable in the statement.

Syntax

```
DBMS_SQL.BIND_VARIABLE (
   c              IN INTEGER,
   name           IN VARCHAR2,
   value          IN <datatype>);
```

Where *datatype* can be any of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
DATE
INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)
NUMBER
TIME(9) (TIME_UNCONSTRAINED)
TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)
VARCHAR2 CHARACTER SET ANY_CS
INTERVAL YEAR TO  MONTH(9) (YMINTERVAL_UNCONSTRAINED)
VARRAY
Nested table
```

Notice that `BIND_VARIABLE` is overloaded to accept different data types.

The following syntax is also supported for `BIND_VARIABLE`. The square brackets `[]` indicate an optional parameter for the `BIND_VARIABLE` function.

```
DBMS_SQL.BIND_VARIABLE (
   c            IN INTEGER,
   name         IN VARCHAR2,
   value        IN VARCHAR2 CHARACTER SET ANY_CS [,out_value_size IN INTEGER]);
```

To bind `CHAR`, `RAW`, and `ROWID` data, you can use the following variations on the following syntax:

```
DBMS_SQL.BIND_VARIABLE_CHAR (
   c              IN INTEGER,
   name           IN VARCHAR2,
   value          IN CHAR CHARACTER SET ANY_CS [,out_value_size IN INTEGER]);

DBMS_SQL.BIND_VARIABLE_RAW (
   c              IN INTEGER,
   name           IN VARCHAR2,
   value          IN RAW [,out_value_size IN INTEGER]);

DBMS_SQL.BIND_VARIABLE_ROWID (
   c              IN INTEGER,
   name           IN VARCHAR2,
   value          IN ROWID);
```

Parameters

Table 6–7 BIND_VARIABLE procedure parameters

Parameter	Description
`c`	ID number of the cursor where the value is to be bound.
`name`	Name of the variable in the statement.
`value`	Value to bind to the variable in the cursor. For `IN` and `IN OUT` variables, the value has the same type as the type of the value being passed in for this parameter.
`out_value_size`	Maximum expected `OUT` value size, in bytes, for the `VARCHAR2`, `RAW`, `CHAR OUT` or `IN OUT` variable. If no size is given, then the length of the current value is used. This parameter must be specified if the *`value`* parameter is not initialized.

Usage notes

If the variable is an `IN` or `IN OUT` variable or an `IN` collection, then the given bind value must be valid for the variable or array type. Bind values for `OUT` variables are ignored.

The bind variables or collections of a SQL statement are identified by their names. When binding a value to a bind variable or bind array, the string identifying it in the statement must contain a leading colon, as shown in the following example:

```
SELECT last_name FROM employees WHERE salary > :X;
```

For this example, the corresponding bind call would look similar to the following:

```
BIND_VARIABLE(cursor_name, ':X', 3500);
```

Or:

```
BIND_VARIABLE (cursor_name, 'X', 3500);
```

The length of the bind variable name should be less than or equal to 30 bytes.

Examples

See "Examples" on page 6-11.

CLOSE_CURSOR procedure

This procedure closes a given cursor.

Syntax

```
DBMS_SQL.CLOSE_CURSOR (
   c    IN OUT INTEGER);
```

Parameters

Table 6–8 CLOSE_CURSOR procedure parameters

Parameter	Description
c	For `IN`, the ID number of the cursor to close. For `OUT`, the cursor is set to `NULL`. After you call `CLOSE_CURSOR`, the memory allocated to the cursor is released and you can no longer fetch from that cursor.

COLUMN_VALUE procedure

This procedure is used to access the data fetched by calling the FETCH_ROWS function. It returns the value of the cursor element for a given position in a given cursor.

Syntax

```
DBMS_SQL.COLUMN_VALUE (
   c                 IN  INTEGER,
   position          IN  INTEGER,
   value             OUT <datatype>
 [,column_error      OUT NUMBER]
 [,actual_length     OUT INTEGER]);
```

Where square brackets `[]` indicate optional parameters and *`datatype`* can be any of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
DATE
INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)
NUMBER
TIME(9) (TIME_UNCONSTRAINED)
TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)
VARCHAR2 CHARACTER SET ANY_CS
INTERVAL YEAR TO  MONTH(9) (YMINTERVAL_UNCONSTRAINED)
VARRAY
Nested table
```

For variables containing `CHAR`, `RAW`, and `ROWID` data, you can use the following variations on the syntax:

```
DBMS_SQL.COLUMN_VALUE_CHAR (
   c               IN  INTEGER,
   position        IN  INTEGER,
   value           OUT CHAR CHARACTER SET ANY_CS
 [,column_error    OUT NUMBER]
 [,actual_length   OUT INTEGER]);

DBMS_SQL.COLUMN_VALUE_RAW (
   c               IN  INTEGER,
   position        IN  INTEGER,
   value           OUT RAW
 [,column_error    OUT NUMBER]
 [,actual_length   OUT INTEGER]);

DBMS_SQL.COLUMN_VALUE_ROWID (
   c               IN  INTEGER,
   position        IN  INTEGER,
   value           OUT ROWID
 [,column_error    OUT NUMBER]
 [,actual_length   OUT INTEGER]);
```

The following syntax enables the `COLUMN_VALUE` procedure to accommodate bulk operations:

```
DBMS_SQL.COLUMN_VALUE(
   c                 IN             INTEGER,
   position          IN             INTEGER,
   <param_name>      IN OUT NOCOPY  <table_type>);
```

Where the *`param_name`* and its corresponding *`table_type`* can be any of these matching pairs:

```
<bdbl_tab>     dbms_sql.Binary_Double_Table
<bflt_tab>     dbms_sql.Binary_Float_Table
<c_tab>        dbms_sql.Varchar2_Table
<d_tab>        dbms_sql.Date_Table
<ids_tab>      dbms_sql.Interval_Day_To_Second_Table
<iym_tab>      dbms_sql.Interval_Year_To_Month_Table
<n_tab>        dbms_sql.Number_Table
<tm_tab>       dbms_sql.Time_Table
<tms_tab>      dbms_sql.Timestamp_Table
```

Parameters

Table 6–9 COLUMN_VALUE procedure parameters (single row)

Parameter	Description
`c`	ID number of the cursor from which you are fetching the values.
`position`	Relative position of the column in the cursor. The first column in a statement has position 1.
`value`	Returns the value at the specified column. Oracle raises exception `ORA-06562`, `inconsistent_type`, if the type of this output parameter differs from the actual type of the value, as defined by the call to `DEFINE_COLUMN`.
`column_error`	Returns any error code for the specified column value.
`actual_length`	The actual length, before any truncation, of the value in the specified column.

Table 6–10 COLUMN_VALUE procedure parameters (bulk)

Parameter	Description
`c`	ID number of the cursor from which you are fetching the values.
`position`	Relative position of the column in the cursor. The first column in a statement has position 1.
`param_name`	Local variable that has been declared *`table_type`*. The *`param_name`* is an `IN OUT NOCOPY` parameter for bulk operations. For bulk operations, the subprogram appends the new elements at the appropriate (implicitly maintained) index. For example, if, when the DEFINE_ARRAY procedure is used, a batch size (the *`cnt`* parameter) of 10 rows was specified and a start index (*`lower_bnd`*) of 1 was specified, then the first call to this subprogram after calling the FETCH_ROWS function will populate elements at index `1..10`, and the next call will populate elements `11..20`, and so on.

Exceptions

`ORA-06562: Type of out argument must match type of column or bind variable`

This exception is raised if the type of the given `OUT` parameter `value` is different from the actual type of the value. This type was the given type when the column was defined by calling `DEFINE_COLUMN`.

Examples

See "Examples" on page 6-11.

DEFINE_ARRAY procedure

This procedure defines the collection into which the row values are fetched, with a FETCH_ROWS function call, for a given column. This procedure lets you do batch fetching of rows from a single `SELECT` statement. A single fetch brings several rows into the PL/SQL aggregate object.

Scalar types for collections

You can declare a local variable as one of the following table-item types, and then fetch any number of rows into it using `DBMS_SQL`. These are the same types you can specify for the `BIND_ARRAY` procedure.

```
TYPE binary_double_table
                    IS TABLE OF BINARY_DOUBLE  INDEX BY BINARY_INTEGER;
TYPE binary_float_table
                    IS TABLE OF BINARY_FLOAT   INDEX BY BINARY_INTEGER;
TYPE date_table     IS TABLE OF DATE           INDEX BY BINARY_INTEGER;
TYPE interval_day_to_second_table
                    IS TABLE OF dsinterval_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE interval_year_to_month_table
                    IS TABLE OF yminterval_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE number_table   IS TABLE OF NUMBER         INDEX BY BINARY_INTEGER;
TYPE time_table     IS TABLE OF time_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE timestamp_table
                    IS TABLE OF timestamp_unconstrained
                                               INDEX BY BINARY_INTEGER;
TYPE varchar2_table IS TABLE OF VARCHAR2(2000) INDEX BY BINARY_INTEGER;
```

Syntax

```
DBMS_SQL.DEFINE_ARRAY (
   c           IN INTEGER,
   position    IN INTEGER,
   <table_variable>    IN <datatype>
   cnt         IN INTEGER,
   lower_bnd   IN INTEGER);
```

Where *`table_variable`* and its corresponding *`datatype`* can be any of the following matching pairs:

```
<bflt_tab>     dbms_sql.Binary_Float_Table
<bdbl_tab>     dbms_sql.Binary_Double_Table
<c_tab>        dbms_sql.Varchar2_Table
<d_tab>        dbms_sql.Date_Table
<n_tab>        dbms_sql.Number_Table
<tm_tab>       dbms_sql.Time_Table
<tms_tab>      dbms_sql.Timestamp_Table
<ids_tab>      dbms_sql.Interval_Day_To_Second_Table
<iym_tab>      dbms_sql.Interval_Year_To_Month_Table
```

Note that `DEFINE_ARRAY` is overloaded to accept different data types.

Parameters

Table 6–11 DEFINE_ARRAY procedure parameters

Parameter	Description
`c`	ID number of the cursor where the array is to be bound.
`position`	Relative position of the column in the array being defined. The first column in a statement has position 1.
`table_variable`	Local variable that has been declared as *`datatype`*.
`cnt`	Number of rows that must be fetched.
`lower_bnd`	Results are copied into the collection, starting at this lower bound index.

Usage notes

The count (*`cnt`*) must be an integer greater than zero. The *`lower_bnd`* can be positive, negative, or zero. A query on which a `DEFINE_ARRAY` call was issued cannot contain array binds.

Exceptions

```
ORA-29253: Invalid count argument passed to procedure dbms_sql.define_array
```

This exception is raised if the count (*`cnt`*) is less than or equal to zero.

Examples

See "Examples" on page 6-11.

DEFINE_COLUMN procedure

This procedure defines a column to be selected from the given cursor. This procedure is only used with `SELECT` cursors.

The column being defined is identified by its relative position in the `SELECT` list of the statement in the given cursor. The type of the `COLUMN` value determines the type of the column being defined.

Syntax

```
DBMS_SQL.DEFINE_COLUMN (
   c              IN INTEGER,
   position       IN INTEGER,
   column         IN <datatype>);
```

Where `datatype` can be any of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
DATE
INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)
NUMBER
TIME(9) (TIME_UNCONSTRAINED)
TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)
INTERVAL YEAR TO  MONTH(9) (YMINTERVAL_UNCONSTRAINED)
VARRAY
Nested table
```

Note that `DEFINE_COLUMN` is overloaded to accept different data types.

The following syntax is also supported for the `DEFINE_COLUMN` procedure:

```
DBMS_SQL.DEFINE_COLUMN (
   c              IN INTEGER,
   position       IN INTEGER,
   column         IN VARCHAR2 CHARACTER SET ANY_CS,
   column_size    IN INTEGER);
```

To define columns with `CHAR`, `RAW`, and `ROWID` data, you can use the following variations on the procedure syntax:

```
DBMS_SQL.DEFINE_COLUMN_CHAR (
   c              IN INTEGER,
   position       IN INTEGER,
   column         IN CHAR CHARACTER SET ANY_CS,
   column_size    IN INTEGER);

DBMS_SQL.DEFINE_COLUMN_RAW (
   c              IN INTEGER,
   position       IN INTEGER,
   column         IN RAW,
   column_size    IN INTEGER);

DBMS_SQL.DEFINE_COLUMN_ROWID (
   c              IN INTEGER,
   position       IN INTEGER,
   column         IN ROWID);
```

Parameters

Table 6–12 DEFINE_COLUMN procedure parameters

Parameter	Description
`c`	ID number of the cursor for the row being defined to be selected.
`position`	Relative position of the column in the row being defined. The first column in a statement has position 1.
`column`	Value of the column being defined. The type of this value determines the type for the column being defined.
`column_size`	Maximum expected size of the column value, in bytes, for columns of type `VARCHAR2`, `CHAR`, and `RAW`.

Examples

See "Examples" on page 6-11.

DESCRIBE_COLUMNS procedure

This procedure describes the columns for a cursor opened and parsed through `DBMS_SQL`.

Syntax

```
DBMS_SQL.DESCRIBE_COLUMNS (
   c              IN  INTEGER,
   col_cnt        OUT INTEGER,
   desc_t         OUT DBMS_SQL.DESC_TAB);

DBMS_SQL.DESCRIBE_COLUMNS (
   c              IN  INTEGER,
   col_cnt        OUT INTEGER,
   desc_t         OUT DBMS_SQL.DESC_REC);
```

Parameters

Table 6–13 DESCRIBE_COLUMNS procedure parameters

Parameter	Description
c	ID number of the cursor for the columns being described.
col_cnt	Number of columns in the select list of the query.
desc_t	Describe table to fill in with the description of each of the columns of the query.

Examples

See "Examples" on page 6-11.

DESCRIBE_COLUMNS2 procedure

This function describes the specified column. This is an alternative to DESCRIBE_COLUMNS procedure.

Syntax

```
DBMS_SQL.DESCRIBE_COLUMNS2 (
   c              IN  INTEGER,
   col_cnt        OUT INTEGER,
   desc_t         OUT DBMS_SQL.DESC_TAB2);

DBMS_SQL.DESCRIBE_COLUMNS2 (
   c              IN  INTEGER,
   col_cnt        OUT INTEGER,
   desc_t         OUT DBMS_SQL.DESC_REC2);
```

Parameters

Table 6–14   DESCRIBE_COLUMNS2 procedure parameters

Parameter	Description
c	ID number of the cursor for the columns being described.
col_cnt	Number of columns in the select list of the query.
desc_t	Describe table to fill in with the description of each of the columns of the query. This table is indexed from one to the number of elements in the select list of the query.

DESCRIBE_COLUMNS3 procedure

This function describes the specified column. This is an alternative to DESCRIBE_COLUMNS procedure.

Syntax

```
DBMS_SQL.DESCRIBE_COLUMNS3 (
   c              IN  INTEGER,
   col_cnt        OUT INTEGER,
   desc_t         OUT DBMS_SQL.DESC_TAB3);

DBMS_SQL.DESCRIBE_COLUMNS3 (
   c              IN  INTEGER,
   col_cnt        OUT INTEGER,
   desc_t         OUT DBMS_SQL.DESC_REC3);
```

Parameters

Table 6–15 DESCRIBE_COLUMNS3 procedure parameters

Parameter	Description
c	ID number of the cursor for the columns being described.
col_cnt	Number of columns in the select list of the query.
desc_t	Describe table to fill in with the description of each of the columns of the query. This table is indexed from one to the number of elements in the select list of the query.

Usage notes

The cursor passed in by the cursor ID has to be opened and parsed, otherwise an error is raised for an invalid cursor ID.

EXECUTE function

This function executes a given cursor. This function accepts the ID number of the cursor and returns the number of rows processed. The return value is only valid for `INSERT`, `UPDATE`, and `DELETE` statements. For other types of statements, including DDL, the return value is undefined and should be ignored.

Syntax

```
DBMS_SQL.EXECUTE (
   c   IN INTEGER)
  RETURN INTEGER;
```

Parameters

Table 6–16 EXECUTE function parameters

Parameter	Description
c	Cursor ID number of the cursor to execute.

Return value

An `INTEGER` value that indicates the number of rows processed.

EXECUTE_AND_FETCH function

This function executes the given cursor and fetches rows. It provides the same functionality as calling `EXECUTE` and then calling `FETCH_ROWS`; however, calling `EXECUTE_AND_FETCH` may reduce the number of network round trips when used against a remote database.

The `EXECUTE_AND_FETCH` function returns the number of rows actually fetched.

Syntax

```
DBMS_SQL.EXECUTE_AND_FETCH (
   c               IN INTEGER,
   exact           IN BOOLEAN DEFAULT FALSE)
  RETURN INTEGER;
```

Parameters

Table 6–17 EXECUTE_AND_FETCH function parameters

Parameter	Description
c	ID number of the cursor to execute and fetch.
exact	Set to `TRUE` to raise an exception if the number of rows actually matching the query differs from 1. Even if an exception is raised, the rows are still fetched and available.

Return value

An `INTEGER` value indicating the number of rows that were fetched.

Exceptions

```
ORA-01422: Exact fetch returns more than requested number of rows
```

This exception is raised if the number of rows matching the query is not 1.

FETCH_ROWS function

This function fetches a row from a given cursor. A DEFINE_ARRAY procedure call defines the collection into which the row values are fetched.

A `FETCH_ROWS` call fetches the specified number of rows, according to the *`cnt`* parameter of the `DEFINE_ARRAY` call. When you fetch the rows, they are copied into `DBMS_SQL` buffers until you execute a COLUMN_VALUE procedure call, for each column, at which time the rows are copied into the table that was passed as an argument to `COLUMN_VALUE`. The rows are placed in positions *`lower_bnd`*, *`lower_bnd`*+1, *`lower_bnd`*+2, and so on, according to the *`lower_bnd`* setting in the `DEFINE_ARRAY` call. While there are still rows coming in, the user keeps issuing `FETCH_ROWS` and `COLUMN_VALUE` calls. You can call `FETCH_ROWS` repeatedly as long as there are rows remaining to be fetched.

The `FETCH_ROWS` function accepts the ID number of the cursor to fetch and returns the number of rows actually fetched.

Syntax

```
DBMS_SQL.FETCH_ROWS (
   c               IN INTEGER)
  RETURN INTEGER;
```

Parameters

Table 6–18 FETCH_ROWS function parameters

Parameter	Description
`c`	ID number of the cursor to fetch.

Return value

An `INTEGER` value indicating the number of rows that were fetched.

Examples

See "Examples" on page 6-11.

IS_OPEN function

This function checks to see if the given cursor is currently open.

Syntax

```
DBMS_SQL.IS_OPEN (
   c              IN INTEGER)
  RETURN BOOLEAN;
```

Parameters

Table 6–19 IS_OPEN function parameters

Parameter	Description
`c`	Cursor ID number of the cursor to check.

Return value

Returns `TRUE` for any cursor number that has been opened but not closed, and `FALSE` for a `NULL` cursor number. Note that the CLOSE_CURSOR procedure nulls out the cursor variable passed to it.

Exceptions

```
ORA-29471 DBMS_SQL access denied
```

This is raised if an invalid cursor ID number is detected. Once a session has encountered and reported this error, every subsequent `DBMS_SQL` call in the same session will raise this error, meaning that `DBMS_SQL` is non-operational for this session.

LAST_ERROR_POSITION function

This function returns the byte offset in the SQL statement text where the error occurred. The first character in the SQL statement is at position 0.

Syntax

```
DBMS_SQL.LAST_ERROR_POSITION
   RETURN INTEGER;
```

Return value

An `INTEGER` value indicating the byte offset in the SQL statement text where the error occurred.

Usage notes

Call this function after a `PARSE` call, before any other `DBMS_SQL` procedures or functions are called.

LAST_ROW_COUNT function

This function returns the cumulative count of the number of rows fetched.

Syntax

```
DBMS_SQL.LAST_ROW_COUNT
   RETURN INTEGER;
```

Return value

An `INTEGER` value indicating the cumulative count of the number of rows that were fetched.

Usage notes

Call this function after a `FETCH_ROWS` or an `EXECUTE_AND_FETCH` call. If called after an `EXECUTE` call, the value returned is zero.

LAST_ROW_ID function

This function returns the rowid of the last row processed.

Because TimesTen does not support rowid of the last row operated on by a DML statement, this function returns `NULL`.

Syntax

```
DBMS_SQL.LAST_ROW_ID
   RETURN ROWID;
```

Return value

Returns `NULL` for TimesTen.

LAST_SQL_FUNCTION_CODE function

This function returns the SQL function code for the statement. These codes are listed in the *Oracle Call Interface Programmer's Guide.*

Syntax

```
DBMS_SQL.LAST_SQL_FUNCTION_CODE
   RETURN INTEGER;
```

Return value

An `INTEGER` value indicating the SQL function code for the statement.

Usage notes

Call this function immediately after the SQL statement is run. Otherwise, the return value is undefined.

OPEN_CURSOR function

This procedure opens a new cursor. The second overload takes a *`security_level`* parameter to apply fine-grained control to the security of the opened cursor. In TimesTen, however, there is no security enforcement: `security_level=0`.

When you no longer need this cursor, you must close it explicitly by calling the CLOSE_CURSOR procedure.

Syntax

```
DBMS_SQL.OPEN_CURSOR
  RETURN INTEGER;

DBMS_SQL.OPEN_CURSOR (
   security_level   IN   INTEGER)
  RETURN INTEGER;
```

Parameters

Table 6–20 OPEN_CURSOR function parameters

Parameter	Description
`security_level`	Specifies the level of security protection to enforce on the opened cursor. Only the security level 0 is valid in TimesTen (levels 1 and 2 are not supported). ▪ Level 0 allows all `DBMS_SQL` operations on the cursor without any security checks. The cursor may be fetched from, and even re-bound and re-executed by, code running with a different effective user ID or roles than at the time the cursor was parsed. This level of security is disabled by default. ▪ Level 1 is not applicable for TimesTen. ▪ Level 2 is not applicable for TimesTen.

Return value

The cursor ID of the new cursor.

Usage notes

You can use cursors to run the same SQL statement repeatedly or to run a new SQL statement. When a cursor is reused, the contents of the corresponding cursor data area are reset when the new SQL statement is parsed. It is never necessary to close and reopen a cursor before reusing it.

PARSE procedure

This procedure parses the given statement in the given cursor. All statements are parsed immediately. In addition, DDL statements are run immediately when parsed.

There are multiple versions of the `PARSE` procedure:

- Taking a `VARCHAR2` statement as an argument
- Taking `VARCHAR2A`, table of `VARCHAR2(32767)`, as an argument. The `VARCHAR2A` overload version of the procedure concatenates elements of a PL/SQL table statement and parses the resulting string. You can use this procedure to parse a statement that is longer than the limit for a single `VARCHAR2` variable by splitting up the statement.

Syntax

```
DBMS_SQL.PARSE (
   c                  IN   INTEGER,
   statement          IN   VARCHAR2,
   language_flag      IN   INTEGER);

DBMS_SQL.PARSE (
   c                  IN   INTEGER,
   statement          IN   DBMS_SQL.VARCHAR2A,
   lb                 IN   INTEGER,
   ub                 IN   INTEGER,
   lfflg              IN   BOOLEAN,
   language_flag      IN   INTEGER);

DBMS_SQL.PARSE (
   c                  IN   INTEGER,
   statement          IN   DBMS_SQL.VARCHAR2S,
   lb                 IN   INTEGER,
   ub                 IN   INTEGER,
   lfflg              IN   BOOLEAN,
   language_flag      IN   INTEGER);
```

Parameters

Table 6–21 PARSE procedure parameters

Parameter	Description
c	ID number of the cursor in which to parse the statement.

statement	SQL statement to be parsed.
	Unlike PL/SQL statements, your SQL statement should not include a final semicolon. For example:
	`DBMS_SQL.PARSE(cursor1, 'BEGIN proc; END;', 2);`
	`DBMS_SQL.PARSE(cursor1, 'INSERT INTO tab VALUES(1)', 2);`

lb	Lower bound for elements in the statement.
ub	Upper bound for elements in the statement.
lfflg	If `TRUE`, insert a linefeed after each element on concatenation.
language_flag	Determines how Oracle handles the SQL statement. For TimesTen, use the `NATIVE` (or 1) setting, which specifies normal behavior for the database to which the program is connected.

Usage notes

- Because client-side code cannot reference remote package variables or constants, you must explicitly use the values of the constants.

  For example, the following code does *not* compile on the client:

  ```
  DBMS_SQL.PARSE(cur_hdl, stmt_str, DBMS_SQL.NATIVE);
  -- uses constant DBMS_SQL.NATIVE
  ```

  The following code works on the client, because the argument is explicitly provided:

  ```
  DBMS_SQL.PARSE(cur_hdl, stmt_str, 1); -- compiles on the client
  ```

- The `VARCHAR2S` type is supported only for backward compatibility. You are advised to use `VARCHAR2A` instead.

Exceptions

```
ORA-24344: Success with compilation error
```

If you create a type, procedure, function, or package that has compilation warnings, this exception is raised but the object is still created.

Examples

See "Examples" on page 6-11.

TO_CURSOR_NUMBER function

This function takes an opened strongly or weakly-typed REF CURSOR and transforms it into a `DBMS_SQL` cursor number.

Syntax

```
DBMS_SQL.TO_CURSOR_NUMBER(
   rc IN OUT SYS_REFCURSOR)
  RETURN INTEGER;
```

Parameters

Table 6–22 TO_CURSOR_NUMBER function parameters

Parameter	Description
rc	REF CURSOR to be transformed into a cursor number.

Return value

Returns a `DBMS_SQL` manageable cursor number transformed from a REF CURSOR.

Usage notes

- The REF CURSOR passed in has to be opened (`OPEN_CURSOR`).
- Once the REF CURSOR is transformed into a `DBMS_SQL` cursor number, the REF CURSOR is no longer accessible by any native dynamic SQL operations.
- Toggling between a REF CURSOR and `DBMS_SQL` cursor number after a fetch has started is not allowed.

Examples

```
CREATE OR REPLACE PROCEDURE DO_QUERY1(sql_stmt VARCHAR2) IS
  TYPE CurType IS REF CURSOR;
  src_cur        CurType;
  curid          NUMBER;
  desctab        DBMS_SQL.DESC_TAB;
  colcnt         NUMBER;
  namevar        VARCHAR2(50);
  numvar         NUMBER;
  datevar        DATE;

BEGIN
    -- sql_stmt := 'select * from employees';
    OPEN src_cur FOR sql_stmt;

    -- Switch from native dynamic SQL to DBMS_SQL
    curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);

    DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);

    -- Define columns
    FOR i IN 1 .. colcnt LOOP
         IF desctab(i).col_type = 2 THEN
           DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
         ELSIF desctab(i).col_type = 12 THEN
            DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
         ELSE
            DBMS_SQL.DEFINE_COLUMN(curid, i, namevar, 25);
         END IF;
    END LOOP;

  -- Fetch Rows
    WHILE DBMS_SQL.FETCH_ROWS(curid) > 0 LOOP
        FOR i IN 1 .. colcnt LOOP
          IF (desctab(i).col_type = 1) THEN
            DBMS_SQL.COLUMN_VALUE(curid, i, namevar);
          ELSIF (desctab(i).col_type = 2) THEN
            DBMS_SQL.COLUMN_VALUE(curid, i, numvar);
          ELSIF (desctab(i).col_type = 12) THEN
            DBMS_SQL.COLUMN_VALUE(curid, i, datevar);
          END IF;
        END LOOP;
    END LOOP;

    DBMS_SQL.CLOSE_CURSOR(curid);
END;
/
```

You could execute this procedure as follows:

```
Command> begin
       > do_query1('select * from employees');
       > end;
       > /

PL/SQL procedure successfully completed.
```

TO_REFCURSOR function

This function takes an opened (`OPEN_CURSOR`), parsed (`PARSE`), and executed (`EXECUTE`) cursor and transforms or migrates it into a PL/SQL-manageable REF CURSOR (a weakly-typed cursor) that can be consumed by PL/SQL native dynamic SQL. This subprogram is only used with `SELECT` cursors.

Syntax

```
DBMS_SQL.TO_REFCURSOR(
   cursor_number IN OUT  INTEGER)
  RETURN SYS_REFCURSOR;
```

Parameters

Table 6–23 TO_REFCURSOR function parameters

Parameter	**Description**
cursor_number	Cursor number of the cursor to be transformed into a REF CURSOR.

Return value

A PL/SQL REF CURSOR transformed from a `DBMS_SQL` cursor number.

Usage notes

- The cursor passed in by the *cursor_number* has to be opened, parsed, and executed. Otherwise an error is raised.
- Once the *cursor_number* is transformed into a REF CURSOR, it is no longer accessible by any `DBMS_SQL` operations.
- After a *cursor_number* is transformed into a REF CURSOR, using `DBMS_SQL.IS_OPEN` results in an error.
- Toggling between REF CURSOR and `DBMS_SQL` cursor number after starting to fetch is not allowed. An error is raised.

Examples

```
CREATE OR REPLACE PROCEDURE DO_QUERY2(mgr_id NUMBER) IS
  TYPE CurType IS REF CURSOR;
  src_cur         CurType;
  curid           NUMBER;
  sql_stmt        VARCHAR2(200);
  ret             INTEGER;
  empnos          DBMS_SQL.Number_Table;
  depts           DBMS_SQL.Number_Table;

BEGIN

  -- DBMS_SQL.OPEN_CURSOR
  curid := DBMS_SQL.OPEN_CURSOR;

  sql_stmt :=
    'SELECT EMPLOYEE_ID, DEPARTMENT_ID from employees where MANAGER_ID = :b1';

  DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.NATIVE);
  DBMS_SQL.BIND_VARIABLE(curid, 'b1', mgr_id);
  ret := DBMS_SQL.EXECUTE(curid);

  -- Switch from DBMS_SQL to native dynamic SQL
  src_cur := DBMS_SQL.TO_REFCURSOR(curid);

  -- Fetch with native dynamic SQL
  FETCH src_cur BULK COLLECT INTO empnos, depts;

  IF empnos.COUNT > 0 THEN
    DBMS_OUTPUT.PUT_LINE('EMPNO DEPTNO');
    DBMS_OUTPUT.PUT_LINE('----- ------');
    -- Loop through the empnos and depts collections
    FOR i IN 1 .. empnos.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE(empnos(i) || '   ' || depts(i));
    END LOOP;
  END IF;

  -- Close cursor
  CLOSE src_cur;
END;
/
```

The following example executes this procedure for a manager ID of 103.

```
Command> begin
       > do_query2(103);
       > end;
       > /
EMPNO DEPTNO
----- ------
104   60
105   60
106   60
107   60

PL/SQL procedure successfully completed.
```

VARIABLE_VALUE procedure

This procedure returns the value of the named variable for a given cursor. It is used to return the values of bind variables inside PL/SQL blocks or of DML statements with a `RETURNING` clause.

Syntax

```
DBMS_SQL.VARIABLE_VALUE (
   c               IN  INTEGER,
   name            IN  VARCHAR2,
   value           OUT NOCOPY <datatype>);
```

Where *`datatype`* can be any of the following types:

```
BINARY_DOUBLE
BINARY_FLOAT
DATE
INTERVAL DAY TO SECOND(9,9) (DSINTERVAL_UNCONSTRAINED)
NUMBER
TIME(9) (TIME_UNCONSTRAINED)
TIMESTAMP(9) (TIMESTAMP_UNCONSTRAINED)
VARCHAR2 CHARACTER SET ANY_CS
INTERVAL YEAR TO  MONTH(9) (YMINTERVAL_UNCONSTRAINED)
VARRAY
Nested table
```

For variables containing `CHAR`, `RAW`, and `ROWID` data, you can use the following variations on the syntax:

```
DBMS_SQL.VARIABLE_VALUE_CHAR (
   c               IN  INTEGER,
   name            IN  VARCHAR2,
   value           OUT CHAR CHARACTER SET ANY_CS);

DBMS_SQL.VARIABLE_VALUE_RAW (
   c               IN  INTEGER,
   name            IN  VARCHAR2,
   value           OUT RAW);

DBMS_SQL.VARIABLE_VALUE_ROWID (
   c               IN  INTEGER,
   name            IN  VARCHAR2,
   value           OUT ROWID);
```

The following syntax enables the `VARIABLE_VALUE` procedure to accommodate bulk operations:

```
DBMS_SQL.VARIABLE_VALUE (
   c                 IN   INTEGER,
   name              IN   VARCHAR2,
   value             OUT NOCOPY <table_type>);
```

For bulk operations, *`table_type`* can be any of the following:

```
dbms_sql.Binary_Double_Table
dbms_sql.Binary_Float_Table
dbms_sql.Date_Table
dbms_sql.Interval_Day_To_Second_Table
dbms_sql.Interval_Year_To_Month_Table
dbms_sql.Number_Table
dbms_sql.Time_Table
dbms_sql.Timestamp_Table
dbms_sql.Varchar2_Table
```

Parameters

Table 6–24 VARIABLE_VALUE procedure parameters

Parameter	Description
c	ID number of the cursor from which to get the values.
name	Name of the variable for which you are retrieving the value.
value	For the single row option, this is the value of the variable for the specified position. For the array option, this is the local variable that has been declared *table_type*. For bulk operations, *value* is an `OUT NOCOPY` parameter.

Exceptions

```
ORA-06562: Type of out argument must match type of column or bind variable
```

This is raised if the type of the output parameter differs from the type of the value as defined by the `BIND_VARIABLE` call.

Examples

See "Examples" on page 6-11.

7

DBMS_UTILITY

The `DBMS_UTILITY` package provides various utility subprograms.

This chapter contains the following topics:

- Using DBMS_UTILITY
  - Security model
  - Constants
  - Types
  - Exceptions
- Summary of DBMS_UTILITY subprograms

Using DBMS_UTILITY

- Security model
- Constants
- Types
- Exceptions

Security model

`DBMS_UTILITY` runs with the privileges of the calling user for the NAME_RESOLVE procedure and the COMPILE_SCHEMA procedure. This is necessary so that the SQL works correctly.

The package does not run as `SYS`.

Constants

The `DBMS_UTILITY` package uses the constants shown in Table 7–1.

Table 7–1 DBMS_UTILITY constants

Name	Type	Value	Description
`INV_ERROR_ON_RESTRICTIONS`	`BINARY_INTEGER`	1	This constant is the only legal value for the *`p_option_flags`* parameter of the `INVALIDATE` subprogram.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

Types

- `dblink_array`
- `index_table_type`
- `instance_record`
- `lname_array`
- `name_array`
- `number_array`
- `uncl_array`

dblink_array

```
TYPE dblink_array IS TABLE OF VARCHAR2(128) INDEX BY BINARY_INTEGER;
```

Lists of database links would be stored here. (TimesTen does not support dblinks.)

index_table_type

```
TYPE index_table_type IS TABLE OF BINARY_INTEGER INDEX BY BINARY_INTEGER;
```

The order in which objects should be generated is returned here.

instance_record

```
TYPE instance_record IS RECORD (
     inst_number   NUMBER,
     inst_name     VARCHAR2(60));
TYPE instance_table IS TABLE OF instance_record INDEX BY BINARY_INTEGER;
```

The list of active instance number and instance name.

The starting index of `instance_table` is 1; `instance_table` is dense.

lname_array

```
TYPE lname_array IS TABLE OF VARCHAR2(4000) index by BINARY_INTEGER;
```

Lists of `long NAME` should be stored here, including fully qualified attribute names.

name_array

```
TYPE name_array IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
```

Lists of *`NAME`* should be stored here.

number_array

```
TYPE number_array IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```

The order in which objects should be generated is returned here.

uncl_array

```
TYPE uncl_array IS TABLE OF VARCHAR2(227) INDEX BY BINARY_INTEGER;
```

Lists of "*USER*"."*NAME*"."*COLUMN*"@`LINK` should be stored here.

Exceptions

The following table lists the exceptions raised by `DBMS_UTILITY`.

Table 7–2 Exceptions Raised by DBMS_UTILITY

Exception	Error Code	Description
`INV_NOT_EXIST_OR_NO_PRIV`	-24237	Raised by the `INVALIDATE` subprogram when the *`object_id`* argument is `NULL` or invalid, or when the caller does not have `CREATE` privilege on the object being invalidated.
`INV_MALFORMED_SETTINGS`	-24238	Raised by the `INVALIDATE` subprogram if a compiler setting is specified more than once in the *`p_plsql_object_settings`* parameter.
`INV_RESTRICTED_OBJECT`	-24239	Raised by the `INVALIDATE` subprogram when different combinations of conditions pertaining to the *`p_object_id`* parameter are contravened.

Summary of DBMS_UTILITY subprograms

Table 7–3 DBMS_UTILITY Package Subprograms

Subprogram	Description
CANONICALIZE procedure	Canonicalizes a given string.
COMMA_TO_TABLE procedure	Converts a comma-delimited list of names into a PL/SQL table of names.
COMPILE_SCHEMA procedure	Compiles all procedures, functions, packages, and views in the specified schema.
DB_VERSION procedure	Returns version information for the database. Returns `NULL` for the compatibility setting because TimesTen does not support the system parameter `COMPATIBLE`.
FORMAT_CALL_STACK function	Formats the current call stack.
FORMAT_ERROR_BACKTRACE function	Formats the backtrace from the point of the current error to the exception handler where the error has been caught.
FORMAT_ERROR_STACK function	Formats the current error stack.
GET_CPU_TIME function	Returns the current CPU time in hundredths of a second.
GET_DEPENDENCY procedure	Shows the dependencies on the object passed in.
GET_ENDIANNESS function	Returns the endianness of your database platform.
GET_HASH_VALUE function	Computes a hash value for the given string.
GET_SQL_HASH function	Computes the hash value for a given string using the MD5 algorithm.
GET_TIME function	Finds out the current time in hundredths of a second.
INVALIDATE procedure	Invalidates a database object and (optionally) modifies its PL/SQL compiler parameter settings.
IS_BIT_SET function	Returns bit setting.
NAME_RESOLVE procedure	Resolves the given name of the form: `[[a.]b.]c[@dblink]` Where *a*, *b*, and *c* are SQL identifiers and *`dblink`* is a dblink. Do not use `@`*`dblink`*. TimesTen does not support database links.
NAME_TOKENIZE procedure	Calls the parser to parse the given name: `'a[.b[.c]][@dblink]"` Where *a*, *b*, and *c* are SQL identifiers and *`dblink`* is a dblink. Strips double quotes or converts to uppercase if there are no quotes. Ignores comments and does not perform semantic analysis. Missing values are `NULL`. Do not use `@`*`dblink`*. TimesTen does not support database links.
TABLE_TO_COMMA procedure	Converts a PL/SQL table of names into a comma-delimited list of names.
VALIDATE procedure	Validates the object described either by owner, name, and namespace or by object ID.

CANONICALIZE procedure

This procedure canonicalizes the given string. The procedure handles a single reserved or key word (such as "`table`"), and strips off white spaces for a single identifier. For example, " `table`" becomes `TABLE`.

Syntax

```
DBMS_UTILITY.CANONICALIZE(
   name          IN    VARCHAR2,
   canon_name    OUT   VARCHAR2,
   canon_len     IN    BINARY_INTEGER);
```

Parameters

Table 7–4 CANONICALIZE procedure parameters

Parameter	Description
`name`	The string to be canonicalized.
`canon_name`	The canonicalized string.
`canon_len`	The length of the string (in bytes) to canonicalize.

Return value

Returns the first *`canon_len`* bytes in *`canon_name`*.

Usage notes

- If *`name`* value is `NULL`, the *`canon_name`* value becomes `NULL`.
- If `name` is not a dotted name, and if `name` begins and ends with a double quote, remove both quotes. Alternatively, convert to upper case with `NLS_UPPER`. Note that this case does not include a name with special characters, such as a space, but is not doubly quoted.
- If name is a dotted name (such as `a."b".c`), for each component in the dotted name in the case in which the component begins and ends with a double quote, no transformation will be performed on this component. Alternatively, convert to upper case with `NLS_UPPER` and apply begin and end double quotes to the capitalized form of this component. In such a case, each canonicalized component will be concatenated in the input position, separated by ".".
- Any other character after `a[.b]*` will be ignored.
- The procedure does not handle cases like `'A B.'`

Examples

- `a` becomes `A`.
- `"a"` becomes `a`.
- `"a".b` becomes `"a"."B"`.
- `"a".b,c.f` becomes `"a"."B"` with`",c.f"` ignored.

COMMA_TO_TABLE procedure

This procedure converts a comma-delimited list of names into a PL/SQL table of names. The second version supports fully qualified attribute names.

Syntax

```
DBMS_UTILITY.COMMA_TO_TABLE (
   list   IN  VARCHAR2,
   tablen OUT BINARY_INTEGER,
   tab    OUT dbms_utility.uncl_array);

DBMS_UTILITY.COMMA_TO_TABLE (
   list   IN  VARCHAR2,
   tablen OUT BINARY_INTEGER,
   tab    OUT dbms_utility.lname_array);
```

Parameters

Table 7–5 COMMA_TO_TABLE procedure parameters

Parameter	**Description**
`list`	Comma-delimited list of names, where a name should have the following format for the first version of the procedure: *`a`*`[.`*`b`*`[.`*`c`*`]][@`*`d`*`]` Or the following format for the second version of the procedure: *`a`*`[.`*`b`*`]*` Where *`a`*, *`b`*, *`c`*, and *`d`* are simple identifiers (quoted or unquoted).
`tablen`	Number of tables in the PL/SQL table.
`tab`	PL/SQL table that contains list of names.

Return value

A PL/SQL table with values `1..`*`n`*, and *`n`*`+1` is `NULL`.

Usage notes

The *`list`* must be a non-empty, comma-delimited list. Anything other than a comma-delimited list is rejected. Commas inside double quotes do not count.

Entries in the comma-delimited list cannot include multibyte characters.

The values in *`tab`* are copied from the original list, with no transformations.

COMPILE_SCHEMA procedure

This procedure compiles all procedures, functions, packages, and views in the specified schema.

Syntax

```
DBMS_UTILITY.COMPILE_SCHEMA (
   schema          IN VARCHAR2,
   compile_all     IN BOOLEAN DEFAULT TRUE,
   reuse_settings  IN BOOLEAN DEFAULT FALSE);
```

Parameters

Table 7–6 COMPILE_SCHEMA procedure parameters

Parameter	Description
schema	Name of the schema.
compile_all	If `TRUE`, compile everything within the schema regardless of whether status is `VALID`. If `FALSE`, compile only objects with status `INVALID`.
reuse_settings	Indicates whether the session settings in the objects should be reused, or whether the current session settings should be adopted instead.

Usage notes

- Note that this subprogram is a wrapper for the RECOMP_SERIAL procedure included with the `UTL_RECOMP` package.
- After calling this procedure, you should select from view `ALL_OBJECTS` for items with status `INVALID` to see if all objects were successfully compiled.
- To see the errors associated with invalid objects, you can use the `ttIsql show errors` command:

```
Command> show errors [{FUNCTION | PROCEDURE | PACKAGE | PACKAGE BODY}
[schema.]name];
```

Examples:

```
Command> show errors function foo;
Command> show errors procedure fred.bar;
Command> show errors package body emp_actions;
```

Exceptions

Table 7–7 COMPILE_SCHEMA procedure exceptions

Exception	Description
`ORA-20000`	Insufficient privileges for some object in this schema.
`ORA-20001`	Cannot recompile `SYS` objects.
`ORA-20002`	Maximum iterations exceeded. Some objects may not have been recompiled.

DB_VERSION procedure

This procedure returns version information for the database.

Returns `NULL` for the compatibility setting because TimesTen does not support the system parameter `COMPATIBLE`.

Syntax

```
DBMS_UTILITY.DB_VERSION (
   version       OUT VARCHAR2,
   compatibility OUT VARCHAR2);
```

Parameters

Table 7–8 DB_VERSION procedure parameters

Parameter	Description
version	A string that represents the internal software version of the database (for example, 7.1.0.0.0). The length of this string is variable and is determined by the database version.
compatibility	The compatibility setting of the database. In TimesTen, `DB_VERSION` returns `NULL` for the compatibility setting because TimesTen does not support the system parameter `COMPATIBLE`.

FORMAT_CALL_STACK function

This function formats the current call stack. It can be used on any stored procedure to access the call stack and is useful for debugging.

Syntax

```
DBMS_UTILITY.FORMAT_CALL_STACK
  RETURN VARCHAR2;
```

Return value

Returns the call stack, up to 2000 bytes.

FORMAT_ERROR_BACKTRACE function

This procedure displays the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output is similar to the output of the `SQLERRM` function, but not subject to the same size limitation.

Syntax

```
DBMS_UTILITY.FORMAT_ERROR_BACKTRACE
  RETURN VARCHAR2;
```

Return value

The backtrace string. A null string is returned if no error is currently being handled.

Examples

Script format_error_backtrace.sql

Execute the following script from `ttIsql`, using the `run` command.

```
CREATE OR REPLACE PROCEDURE Log_Errors ( i_buff in varchar2 ) IS
  g_start_pos integer := 1;
  g_end_pos  integer;

  FUNCTION Output_One_Line RETURN BOOLEAN IS
  BEGIN
    g_end_pos := Instr ( i_buff, Chr(10), g_start_pos );

    CASE g_end_pos > 0
      WHEN true THEN
        DBMS_OUTPUT.PUT_LINE ( Substr ( i_buff, g_start_pos,
                               g_end_pos-g_start_pos ) );
        g_start_pos := g_end_pos+1;
        RETURN TRUE;

      WHEN FALSE THEN
        DBMS_OUTPUT.PUT_LINE ( Substr ( i_buff, g_start_pos,
                               (Length(i_buff)-g_start_pos)+1 ) );
        RETURN FALSE;
    END CASE;
  END Output_One_Line;

BEGIN
  WHILE Output_One_Line() LOOP NULL;
  END LOOP;
END Log_Errors;
/

CREATE OR REPLACE PROCEDURE P0 IS
  e_01476 EXCEPTION; pragma exception_init ( e_01476, -1476 );
BEGIN
  RAISE e_01476;
END P0;
/
Show Errors
```

```
CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
  P0();
END P1;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
  P1();
END P2;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
  P2();
END P3;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE P4 IS
BEGIN
  P3();
END P4;
/
CREATE OR REPLACE PROCEDURE P5 IS
BEGIN
  P4();
END P5;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE Top_Naive IS
BEGIN
  P5();
END Top_Naive;
/
SHOW ERRORS

CREATE OR REPLACE PROCEDURE Top_With_Logging IS
  -- NOTE: SqlErrm in principle gives the same info as Format_Error_Stack.
  -- But SqlErrm is subject to some length limits,
  -- while Format_Error_Stack is not.
BEGIN
  P5();
EXCEPTION
  WHEN OTHERS THEN
    Log_Errors ( 'Error_Stack...' || Chr(10) ||
      DBMS_UTILITY.FORMAT_ERROR_STACK() );
    Log_Errors ( 'Error_Backtrace...' || Chr(10) ||
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '----------' );
END Top_With_Logging;
/
SHOW ERRORS
```

Execute Top_Naive

This shows the results of executing the `Top_Naive` procedure that is created in the script, assuming user `SCOTT` ran the script and executed the procedure.

```
Command> set serveroutput on
Command> begin
       > Top_Naive();
       > end;
       > /
 8507: ORA-01476: divisor is equal to zero
 8507: ORA-06512: at "SCOTT.P0", line 4
 8507: ORA-06512: at "SCOTT.P1", line 3
 8507: ORA-06512: at "SCOTT.P2", line 3
 8507: ORA-06512: at "SCOTT.P3", line 3
 8507: ORA-06512: at "SCOTT.P4", line 3
 8507: ORA-06512: at "SCOTT.P5", line 3
 8507: ORA-06512: at "SCOTT.TOP_NAIVE", line 3
 8507: ORA-06512: at line 2
The command failed.
```

This output shows the call stack at the point where an exception was raised. It shows the backtrace error message as the call stack unwound, starting at the unhandled exception `ORA-01476` raised at `SCOTT.P0` line 4, back to `SCOTT.Top_Naive` line 3.

Execute Top_With_Logging

This shows the results of executing the `Top_With_Logging()` procedure that is created in the script, assuming user `SCOTT` ran the script and executed the procedure.

```
Command> begin
       > Top_With_Logging();
       > end;
       > /
Error_Stack...
ORA-01476: divisor is equal to zero
Error_Backtrace...
ORA-06512: at "SCOTT.P0", line 4
ORA-06512: at "SCOTT.P1", line 3
ORA-06512: at "SCOTT.P2", line 3
ORA-06512: at "SCOTT.P3", line 3
ORA-06512: at "SCOTT.P4", line 3
ORA-06512: at "SCOTT.P5", line 3
ORA-06512: at "SCOTT.TOP_WITH_LOGGING", line 6
----------

PL/SQL procedure successfully completed.
```

This output shows the call stack at the point where an exception was raised. It shows the backtrace error message as the call stack unwound, starting at the unhandled exception `ORA-01476` raised at `SCOTT.P0` line 4, back to `SCOTT.Top_With_Logging` line 6.

ORA-06512 information

Oracle Database Error Messages provides the following information about the `ORA-06512` error:

```
ORA-06512: at stringline string
    Cause: Backtrace message as the stack is unwound by unhandled exceptions.
    Action: Fix the problem causing the exception or write an exception handler
 for this condition. Or you may need to contact your application administrator or
 DBA.
```

FORMAT_ERROR_STACK function

This function formats the current error stack. It can be used in exception handlers to look at the full error stack.

Syntax

```
DBMS_UTILITY.FORMAT_ERROR_STACK
  RETURN VARCHAR2;
```

Return value

This returns the error stack, up to 2000 bytes. A null string is returned if no error is currently being handled.

GET_CPU_TIME function

This function returns the current CPU time in hundredths of a second from some arbitrary epoch.

Syntax

```
DBMS_UTILITY.GET_CPU_TIME
  RETURN NUMBER;
```

Return value

The number of hundredths of a second from some arbitrary epoch.

Usage notes

This subprogram reports cycles (CPU time) used in performing work and is unrelated to clock time or any other fixed reference. Since the base line is rubricator, and the relationship between work performed and the number generated is operating system specific, the amount of work performed is calculated by measuring the difference between a start point and end point for a particular operation.

GET_DEPENDENCY procedure

This procedure shows the dependencies on the object passed in.

Syntax

```
DBMS_UTILITY.GET_DEPENDENCY
  type      IN     VARCHAR2,
  schema    IN     VARCHAR2,
  name      IN     VARCHAR2);
```

Parameters

Table 7–9 GET_DEPENDENCY procedure parameters

Parameter	Description
type	The type of the object. For example, if the object is a table, give the type as "TABLE".
schema	The schema name of the object.
name	The name of the object.

Usage notes

This procedure uses the `DBMS_OUTPUT` package to display results, so you must declare `SET SERVEROUTPUT ON` to view dependencies. Alternatively, any application that checks the `DBMS_OUTPUT` output buffers can invoke this subprogram and then retrieve the output through `DBMS_OUTPUT` subprograms such as `GET_LINES`.

GET_ENDIANNESS function

This function indicates the endianness of the database platform.

Syntax

```
DBMS_UTILITY.GET_ENDIANNESS
 RETURN NUMBER;
```

Return value

A `NUMBER` value indicating the endianness of the database platform: 1 for big-endian or 2 for little-endian.

GET_HASH_VALUE function

This function computes a hash value for the given string.

Syntax

```
DBMS_UTILITY.GET_HASH_VALUE (
   name      IN  VARCHAR2,
   base      IN  NUMBER,
   hash_size IN  NUMBER)
  RETURN NUMBER;
```

Parameters

Table 7–10 GET_HASH_VALUE function parameters

Parameter	Description
name	String to be hashed.
base	Base value where the returned hash value is to start.
hash_size	Desired size of the hash table.

Return value

A hash value based on the input string. For example, to get a hash value on a string where the hash value should be between 1000 and 3047, use 1000 as the base value and 2048 as the *hash_size* value. Using a power of 2 for *hash_size* works best.

GET_SQL_HASH function

This function computes a hash value for the given string using the MD5 algorithm.

Syntax

```
DBMS_UTILITY.GET_SQL_HASH (
   name          IN   VARCHAR2,
  [hash          OUT  RAW,
   pre10ihash    OUT  NUMBER])
  RETURN NUMBER;
```

Parameters

Table 7–11 GET_SQL_HASH procedure parameters

Parameter	Description
name	String to be hashed.
hash	An optional field to store all 16 bytes of returned hash value.
pre10ihash	An optional field to store a pre-Oracle 10*g* database version hash value.

Return value

A hash value (last four bytes) based on the input string. The MD5 hash algorithm computes a 16-byte hash value, but TimesTen returns only the last four bytes to return an actual number. One could use an optional `RAW` parameter to get all 16 bytes and to store the pre-Oracle 10*g* hash value of four bytes in the *`pre10ihash`* optional parameter.

GET_TIME function

This function determines the current time in hundredths of a second and is primarily used for determining elapsed time. The subprogram is called twice, at the beginning and end of a process. The first (earlier) number is subtracted from the second (later) number to determine the time elapsed.

Syntax

```
DBMS_UTILITY.GET_TIME
  RETURN NUMBER;
```

Return value

The number of hundredths of a second from the time at which the subprogram is invoked.

Usage notes

Numbers are returned in the range -2,147,483,648 to 2,147,483,647 depending on platform and system, and your application must take the sign of the number into account in determining the interval. For example, for two negative numbers, application logic must allow that the first (earlier) number will be larger than the second (later) number that is closer to zero. By the same token, your application should also allow for the first (earlier) number to be negative and the second (later) number to be positive.

INVALIDATE procedure

This procedure invalidates a database object and (optionally) modifies its PL/SQL compiler parameter settings. It also invalidates any objects that directly or indirectly depend on the object being invalidated.

Syntax

```
DBMS_UTILITY.INVALIDATE (
   p_object_id             IN  NUMBER,
  [p_plsql_object_settings  IN  VARCHAR2 DEFAULT NULL,
   p_option_flags           BINARY_INTEGER DEFAULT 0]);
```

Parameters

Table 7–12 INVALIDATE procedure parameters

Parameter	Description
`p_object_id`	ID number of object to be invalidated. This equals the value of the `OBJECT_ID` column from `ALL_OBJECTS`. If the *`p_object_id`* argument is `NULL` or invalid then the exception `inv_not_exist_or_no_priv` is raised. The caller of this procedure must have `CREATE` privilege on the object being invalidated, otherwise the `inv_not_exist_or_no_priv` exception is raised.
`p_plsql_object_settings`	This optional parameter is ignored if the object specified by *`p_object_id`* is not a PL/SQL object. If no value is specified for this parameter, the PL/SQL compiler settings are left unchanged, equivalent to `REUSE SETTINGS`. If a value is provided, it must specify the values of the PL/SQL compiler settings separated by one or more spaces. If a setting is specified more than once, the `inv_malformed_settings` exception is raised. The setting values are changed only for the object specified by *`p_object_id`* and do not affect dependent objects that may be invalidated. The setting names and values are case insensitive. If a setting is omitted and `REUSE SETTINGS` is specified, then if a value was specified for the compiler setting in an earlier compilation of this library unit, TimesTen uses that value. If a setting is omitted and `REUSE SETTINGS` was not specified or no value was specified for the parameter in an earlier compilation, then the database will obtain the value for that setting from the session environment.
`p_option_flags`	This parameter is optional and defaults to zero (no flags). Option flags supported by invalidate. ■ `inv_error_on_restrictions` (see "Constants" on page 7-4): The subprogram imposes various restrictions on the objects that can be invalidated. For example, the object specified by *`p_object_id`* cannot be a table. By default, invalidate quietly returns on these conditions (and does not raise an exception). If the caller sets this flag, the exception `inv_restricted_object` is raised.

Usage notes

The object type (`object_type` column from `ALL_OBJECTS`) of the object specified by *`p_object_id`* must be a `PROCEDURE`, `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `LIBRARY`, `OPERATOR`, or `SYNONYM`. If the object is not one of these types and the flag `inv_error_on_restrictions` is specified in *`p_option_flags`*, then the exception `inv_restricted_object` is raised. If `inv_error_on_restrictions` is not specified in this situation, then no action is taken.

If the object specified by *`p_object_id`* is the package specification of `STANDARD` or `DBMS_STANDARD`, or the specification or body of `DBMS_UTILITY`, and if the flag `inv_error_on_restrictions` is specified in *`p_option_flags`*, then the exception `inv_restricted_object` is raised. If `inv_error_on_restrictions` is not specified in this situation, then no action is taken.

If the object specified by *`p_object_id`* is an object type specification and there are tables that depend on the type, and if the flag `inv_error_on_restrictions` is specified in *`p_option_flags`*, then the exception `inv_restricted_object` is raised. If `inv_error_on_restrictions` is not specified, then no action is taken.

Exceptions

Table 7–13 INVALIDATE exceptions

Exception	Description
`INV_NOT_EXIST_OR_NO_PRIV`	Raised when the *`object_id`* argument value is `NULL` or invalid, or when the caller does not have `CREATE` privilege on the object being invalidated.
`INV_MALFORMED_SETTINGS`	Raised if a compiler setting is specified more than once in the *`p_plsql_object_settings`* parameter.
`INV_RESTRICTED_OBJECT`	Raised when different combinations of conditions pertaining to the *`p_object_id`* parameter are contravened.

Examples

This example invalidates a procedure created in the example in "FORMAT_ERROR_BACKTRACE function" on page 7-13. From examining `user_objects`, you can see information for the procedures created in that example. The following describes `user_objects` then queries its contents.

```
Command> describe user_objects;

View SYS.USER_OBJECTS:
  Columns:
    OBJECT_NAME                     VARCHAR2 (30) INLINE
    SUBOBJECT_NAME                  VARCHAR2 (30) INLINE
    OBJECT_ID                       TT_BIGINT NOT NULL
    DATA_OBJECT_ID                  TT_BIGINT
    OBJECT_TYPE                     VARCHAR2 (17) INLINE NOT NULL
    CREATED                         DATE NOT NULL
    LAST_DDL_TIME                   DATE NOT NULL
    TIMESTAMP                       VARCHAR2 (78) INLINE NOT NULL
    STATUS                          VARCHAR2 (7) INLINE NOT NULL
    TEMPORARY                       VARCHAR2 (1) INLINE NOT NULL
    GENERATED                       VARCHAR2 (1) INLINE NOT NULL
    SECONDARY                       VARCHAR2 (1) INLINE NOT NULL
    NAMESPACE                       TT_INTEGER NOT NULL
    EDITION_NAME                    VARCHAR2 (30) INLINE

1 view found.

Command> select * from user_objects;
...
< LOG_ERRORS, <NULL>, 296, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12
:58:22, 2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
```

```
< P0, <NULL>, 297, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P1, <NULL>, 298, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P2, <NULL>, 299, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P3, <NULL>, 300, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P4, <NULL>, 301, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< P5, <NULL>, 302, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:
58:22, 2009-09-18:12:58:22, VALID, N, N, N, 1, <NULL> >
< TOP_WITH_LOGGING, <NULL>, 304, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09
-18 15:19:16, 2009-09-18:15:19:16, VALID, N, N, N, 1, <NULL> >
...
20 rows found.
```

To invalidate the `P5` procedure, for example, specify `object_id` 302 in the `INVALIDATE` call:

```
Command> begin
       > dbms_utility.invalidate(302, 'PLSQL_OPTIMIZE_LEVEL=2 REUSE SETTINGS');
       > end;
       > /
```

This will mark the `P5` procedure invalid and set its `PLSQL_OPTIMIZE_LEVEL` compiler setting to 2. The values of other compiler settings will remain unchanged because `REUSE SETTINGS` is specified. Note that in addition to `P5` being invalidated, any PL/SQL objects that refer to that object are invalidated. Given that `Top_With_Logging` and `Top_Naive` call `P5`, here are the results of the `INVALIDATE` call, querying for all user objects that are now invalid:

```
Command> select * from user_objects where status='INVALID';
< P5, <NULL>, 302, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:
58:22, 2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_WITH_LOGGING, <NULL>, 304, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09
-18 15:19:16, 2009-09-18:15:19:16, INVALID, N, N, N, 1, <NULL> >
3 rows found.
```

A user can explicitly recompile and revalidate an object by calling the `VALIDATE` procedure discussed later in this chapter, or by executing `ALTER PROCEDURE`, `ALTER FUNCTION`, or `ALTER PACKAGE`, as applicable, on the object. Alternatively, each object is recompiled and revalidated automatically the next time it is executed.

IS_BIT_SET function

This function checks the bit setting for the given bit in the given `RAW` value.

Syntax

```
DBMS_UTILITY.IS_BIT_SET (
   r       IN RAW,
   n       IN NUMBER)
  RETURN NUMBER;
```

Parameters

Table 7–14 IS_BIT_SET procedure parameters

Parameter	Description
r	Source raw.
n	Which bit in *r* to check.

Return value

This function returns 1 if bit *n* in `RAW` *r* is set. Bits are numbered high to low with the lowest bit being bit number 1.

NAME_RESOLVE procedure

This procedure resolves the given name of the form:

`[[a.]b.]c[@dblink]`

Where *a*, *b*, and *c* are SQL identifiers and *dblink* is a dblink, including synonym translation and authorization checking as necessary.

Do not use @*dblink*. TimesTen does not support database links.

Syntax

```
DBMS_UTILITY.NAME_RESOLVE (
   name          IN  VARCHAR2,
   context       IN  NUMBER,
   schema        OUT VARCHAR2,
   part1         OUT VARCHAR2,
   part2         OUT VARCHAR2,
   dblink        OUT VARCHAR2,
   part1_type    OUT NUMBER,
   object_number OUT NUMBER);
```

Parameters

Table 7–15 NAME_RESOLVE procedure parameters

Parameter	Description
name	Name of the object. This can be of the form: `[[a.]b.]c[@dblink]` Where *a*, *b*, and *c* are SQL identifiers and *dblink* is a dblink. TimesTen does not support dblinks. No syntax checking is performed on the dblink. If a dblink is specified, or if the name resolves to something with a dblink, then the object is not resolved, but the *schema*, *part1*, *part2*, and *dblink* `OUT` parameters receive values. The *a*, *b* and *c* entries may be delimited identifiers, and may contain Globalization Support (NLS) characters, either single or multibyte.
context	Must be an integer between 0 and 9. ■ 0 - table ■ 1 - PL/SQL (for two-part names) ■ 2 - sequences ■ 3 - trigger (not applicable for TimesTen) ■ 4 - Java source (not applicable for TimesTen) ■ 5 - Java resource (not applicable for TimesTen) ■ 6 - Java class (not applicable for TimesTen) ■ 7 - type (not applicable for TimesTen) ■ 8 - Java shared data (not applicable for TimesTen) ■ 9 - index
schema	Schema of the object: *c*. If no schema is specified in *name*, then *schema* is determined by resolving the name.
part1	First part of the name. The type of this name is specified *part1_type* (synonym or package).

Table 7–15 (Cont.) NAME_RESOLVE procedure parameters

Parameter	Description
`part2`	If this is non-null, then this is a subprogram name. If *`part1`* is non-null, then the subprogram is within the package indicated by *`part1`*. If *`part1`* is null, the subprogram is a top-level subprogram.
`dblink`	If this is non-null, then a database link was specified as part of *`name`*, or *`name`* was a synonym that resolved to something with a database link. In this case, if further name translation is desired, you must call `NAME_RESOLVE` on this remote node. TimesTen does not support database links.
`part1_type`	Type of *`part1`* is: ▪ 5 - synonym ▪ 7 - procedure (top level) ▪ 8 - function (top level) ▪ 9 - package
`object_number`	Object identifier.

Exceptions

All errors are handled by raising exceptions. A wide variety of exceptions are possible, based on the various syntax errors that are possible when specifying object names.

NAME_TOKENIZE procedure

This procedure calls the parser to parse the input name as:

`"a[.b[.c]][@dblink]"`

Where *`a`*, *`b`*, and *`c`* are SQL identifiers and *`dblink`* is a dblink. It strips double quotes, or converts to uppercase if there are no quotes. It ignores comments of all sorts, and does no semantic analysis. Missing values are left as `NULL`.

Do not use `@`*`dblink`*. TimesTen does not support database links.

Syntax

```
DBMS_UTILITY.NAME_TOKENIZE (
   name    IN  VARCHAR2,
   a       OUT VARCHAR2,
   b       OUT VARCHAR2,
   c       OUT VARCHAR2,
   dblink  OUT VARCHAR2,
   nextpos OUT BINARY_INTEGER);
```

Parameters

Table 7–16   NAME_TOKENIZE procedure parameters

Parameter	Description
name	The input name, consisting of SQL identifiers (for example, `scott.foo`).
a	Output for the first token of the name.
b	Output for the second token of the name (if applicable).
c	Output for the third token of the name (if applicable).
dblink	Output for the dblink of the name. This is not used in TimesTen.
nextpos	Next position after parsing the input name.

Examples

Consider the following script:

```
declare
   a varchar2(30);
   b varchar2(30);
   c varchar2(30);
   d varchar2(30);
   next integer;
begin
   dbms_utility.name_tokenize('scott.foo', a, b, c, d, next);
   dbms_output.put_line('a: ' || a);
   dbms_output.put_line('b: ' || b);
   dbms_output.put_line('c: ' || c);
   dbms_output.put_line('d: ' || d);
   dbms_output.put_line('next: ' || next);
end;
/
```

This produces the following output.

```
a: SCOTT
b: FOO
c:
d:
next: 9

PL/SQL procedure successfully completed.
```

TABLE_TO_COMMA procedure

This procedure converts a PL/SQL table of names into a comma-delimited list of names. This takes a PL/SQL table, 1..*n*, terminated with *n*+1 being `NULL`. The second version supports fully qualified attribute names.

Syntax

```
DBMS_UTILITY.TABLE_TO_COMMA (
   tab    IN  dbms_utility.uncl_array,
   tablen OUT BINARY_INTEGER,
   list   OUT VARCHAR2);

DBMS_UTILITY.TABLE_TO_COMMA (
   tab    IN  dbms_utility.lname_array,
   tablen OUT BINARY_INTEGER,
   list   OUT VARCHAR2);
```

Parameters

Table 7–17 TABLE_TO_COMMA procedure parameters

Parameter	Description
tab	PL/SQL table that contains list of table names.
tablen	Number of tables in the PL/SQL table.
list	Comma-delimited list of tables.

Return value

A `VARCHAR2` value with a comma-delimited list and the number of elements found in the table.

VALIDATE procedure

Validates the object described either by owner, name, and namespace or by object ID.

Syntax

```
DBMS_UTILITY.VALIDATE(
    object_id     IN  NUMBER);

DBMS_UTILITY.VALIDATE(
   owner          IN  VARCHAR2,
   objname        IN  VARCHAR2,
   namespace      NUMBER,
   edition_name   VARCHAR2 := NULL;
```

Parameters

Table 7–18 VALIDATE procedure parameters

Parameter	Description
owner	Name of the user who owns the object. Same as the `OWNER` field in `ALL_OBJECTS`.
objname	Name of the object to be validated. Same as the `OBJECT_NAME` field in `ALL_OBJECTS`.
namespace	Namespace of the object. Same as the *`namespace`* field in `obj$`. Equivalent numeric values are as follows: ▪ 1 - table or procedure ▪ 2 - body ▪ 3 - trigger (not applicable for TimesTen) ▪ 4 - index ▪ 5 - cluster ▪ 8 - LOB (not applicable for TimesTen) ▪ 9 - directory ▪ 10 - queue ▪ 11 - replication object group ▪ 12 - replication propagator ▪ 13 - Java source (not applicable for TimesTen ▪ 14 - Java resource (not applicable for TimesTen) ▪ 58 - model (data mining)
edition_name	Reserved for future use.

Usage notes

- Executing `VALIDATE` on a subprogram will also validate subprograms that it references. (See the example below.)
- No errors are raised if the object does not exist, is already valid, or is an object that cannot be validated.
- The INVALIDATE procedure invalidates a database object and optionally changes its PL/SQL compiler parameter settings. The object to be invalidated is specified by its `object_id` value.

Examples

This example starts where the `INVALIDATE` example in "INVALIDATE procedure" on page 7-23 left off. Assume `P5`, `Top_Naive`, and `Top_With_Logging` are invalid, shown as follows:

```
Command> select * from user_objects where status='INVALID';
< P5, <NULL>, 302, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:58:22,
2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-18 12:
58:22, 2009-09-18:12:58:22, INVALID, N, N, N, 1, <NULL> >
< TOP_WITH_LOGGING, <NULL>, 304, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09
-18 15:19:16, 2009-09-18:15:19:16, INVALID, N, N, N, 1, <NULL> >
3 rows found.
```

Validating `Top_With_Logging`, for example, will also validate `P5`, because it calls `P5` (leaving only `Top_Naive` invalid):

```
Command> begin
       > dbms_utility.validate(304);
       > end;
       > /

PL/SQL procedure successfully completed.

Command> select * from user_objects where status='INVALID';
< TOP_NAIVE, <NULL>, 303, <NULL>, PROCEDURE, 2009-09-18 12:53:45, 2009-09-21 11:
14:37, 2009-09-21:11:14:37, INVALID, N, N, N, 1, <NULL> >
1 row found.
```

8

TT_DB_VERSION

The `TT_DB_VERSION` package indicates the TimesTen version numbers.

This chapter contains the following topics:

- Using TT_DB_VERSION
    - Overview
    - Constants
    - Examples

Using TT_DB_VERSION

- Overview
- Constants
- Examples

Overview

The `TT_DB_VERSION` package indicates the TimesTen major release (`VERSION`) and minor release (`RELEASE`) version numbers. For any TimesTen 11.2.1.x release, the `VERSION` value is 1121. Depending on the minor release, the `RELEASE` value would be 0, 1, 2, and so on.

The package for Oracle TimesTen In-Memory Database Release 11.2.1.9.0 is as follows:

```
PACKAGE TT_DB_VERSION IS
   VERSION CONSTANT PLS_INTEGER := 1121; -- major release number
   RELEASE CONSTANT PLS_INTEGER := 9;  -- minor release number
END TT_DB_VERSION;
```

Constants

The `TT_DB_VERSION` package contains different constant values for different TimesTen releases. The Oracle TimesTen In-Memory Database Release 11.2.1.9.0 version of the `TT_DB_VERSION` package has the values shown in Table 8–1.

Table 8–1 TT_DB_VERSION constants

Name	Type	Value	Description
`VERSION`	`BINARY_INTEGER`	1121	Current major release of the Oracle TimesTen In-Memory Database.
`RELEASE`	`BINARY_INTEGER`	9	Current minor release of the Oracle TimesTen In-Memory Database.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

Examples

See "Examples" on page 10-6 in the `UTL_IDENT` chapter for an example that uses both that package and `TT_DB_VERSION` for conditional compilation.

9

UTL_FILE

With the `UTL_FILE` package, PL/SQL programs can read and write operating system text files. `UTL_FILE` provides a restricted version of operating system stream file I/O.

This chapter contains the following topics:

- Using UTL_FILE
  - Security model
  - Rules and limits
  - Exceptions
  - Examples
- Data structures
- Summary of UTL_FILE subprograms

Using UTL_FILE

- Security model
- Operational notes
- Rules and limits
- Exceptions
- Examples

Security model

In TimesTen 11.2.x releases, `UTL_FILE` is limited to the directory *`install_dir`*`/plsql/utl_file_temp`. Access does not extend to subdirectories of this directory. In addition, access is subject to file system permission checking. The instance administrator can grant `UTL_FILE` access to specific users as desired. Users can reference this `UTL_FILE` directory by using the string 'UTL_FILE_TEMP' for the location parameter in `UTL_FILE` subprograms. This predefined string is used in the same way as directory object names in Oracle Database.

You cannot use `UTL_FILE` with a link, which could be used to circumvent desired access limitations. Specifying a link as the file name will cause `FOPEN` to fail with an error.

In TimesTen direct mode, the application owner is owner of the file. In client/server mode, the server owner is owner of the file.

`UTL_FILE_DIR` access is not supported in TimesTen.

Important:

- Users do not have execute permission on `UTL_FILE` by default. To use `UTL_FILE` in TimesTen, an `ADMIN` user or instance administrator must explicitly `GRANT EXECUTE` permission on it, such as in the following example:

  ```
  GRANT EXECUTE ON SYS.UTL_FILE TO scott;
  ```

- The privileges needed to access files are operating system specific. `UTL_FILE` privileges give you read and write access to all files within the `UTL_FILE` directory, but not in subdirectories.
- Attempting to apply invalid `UTL_FILE` options will cause unpredictable results.

Operational notes

The file location and file name parameters are supplied to the `FOPEN` function as separate strings, so that the file location can be checked against the `utl_file_temp` directory. Together, the file location and name must represent a legal file name on the system, and the directory must be accessible. Any subdirectories of `utl_file_temp` are not accessible.

`UTL_FILE` implicitly interprets line terminators on read requests, thereby affecting the number of bytes returned on a `GET_LINE` call. For example, the *`len`* parameter of `UTL_FILE.GET_LINE` specifies the requested number of bytes of character data. The number of bytes actually returned to the user will be the least of the following:

- `GET_LINE` *`len`* parameter value
- Number of bytes until the next line terminator character
- The *`max_linesize`* parameter value specified by `UTL_FILE.FOPEN`

The `FOPEN` *`max_linesize`* parameter must be a number in the range 1 and 32767. If unspecified, TimesTen supplies a default value of 1024. The `GET_LINE` *`len`* parameter must be a number in the range 1 and 32767. If unspecified, TimesTen supplies the default value of *`max_linesize`*. If *`max_linesize`* and *`len`* are defined to be different values, then the lesser value takes precedence.

When data encoded in one character set is read and Globalization Support is informed (such as through `NLS_LANG`) that it is encoded in another character set, the result is indeterminate. If `NLS_LANG` is set, it should be the same as the database character set.

Rules and limits

Operating system-specific parameters, such as C-shell environment variables under UNIX, cannot be used in the file location or file name parameters.

`UTL_FILE` I/O capabilities are similar to standard operating system stream file I/O (`OPEN`, `GET`, `PUT`, `CLOSE`) capabilities, but with some limitations. For example, call the `FOPEN` function to return a file handle, which you use in subsequent calls to `GET_LINE` or `PUT` to perform stream I/O to a file. When file I/O is done, call `FCLOSE` to complete any output and free resources associated with the file.

Exceptions

This section describes exceptions that are thrown by `UTL_FILE` subprograms.

> **Note:** In addition to the exceptions listed here, procedures and functions in `UTL_FILE` can raise predefined PL/SQL exceptions such as `NO_DATA_FOUND` or `VALUE_ERROR`. Refer to "Predefined Exceptions" in *Oracle Database PL/SQL Language Reference* for information about those.

Table 9–1 UTL_FILE package exceptions

Exception Name	Description
`ACCESS_DENIED`	Permission to access to the file location is denied.
`CHARSETMISMATCH`	A file is opened using `FOPEN_NCHAR`, but later I/O operations use non-`NCHAR` procedures such as `PUTF` or `GET_LINE`. Or a file is opened using `FOPEN`, but later I/O operations use `NCHAR` functions such as `PUTF_NCHAR` or `GET_LINE_NCHAR`.
`DELETE_FAILED`	The requested file delete operation failed.
`FILE_OPEN`	The requested operation failed because the file is open.
`INTERNAL_ERROR`	Unspecified PL/SQL error.
`INVALID_FILEHANDLE`	File handle is invalid.
`INVALID_FILENAME`	The *`filename`* parameter is invalid.
`INVALID_MAXLINESIZE`	The *`max_linesize`* value for `FOPEN` is out of range. It should be within the range 1 to 32767.
`INVALID_MODE`	The *`open_mode`* parameter in `FOPEN` is invalid.
`INVALID_OFFSET`	Causes of the `INVALID_OFFSET` exception. One of the following: ■ `ABSOLUTE_OFFSET` is `NULL` and `RELATIVE_OFFSET` is `NULL`. ■ `ABSOLUTE_OFFSET` is less than 0. ■ Either offset caused a seek past the end of the file.
`INVALID_OPERATION`	File could not be opened or operated on as requested.
`INVALID_PATH`	File location or name is invalid.
`LENGTH_MISMATCH`	Length mismatch for `CHAR` or `RAW` data.
`READ_ERROR`	Operating system error occurred during the read operation.
`RENAME_FAILED`	The requested file rename operation failed.
`WRITE_ERROR`	Operating system error occurred during the write operation.

Examples

Example 1

This example reads from a file using the `GET_LINE` procedure.

```
DECLARE
  V1 VARCHAR2(32767);
  F1 UTL_FILE.FILE_TYPE;
BEGIN
  -- In this example MAX_LINESIZE is less than GET_LINE's length request
  -- so number of bytes returned will be 256 or less if a line terminator is seen.
  F1 := UTL_FILE.FOPEN('UTL_FILE_TEMP','u12345.tmp','R',256);
  UTL_FILE.GET_LINE(F1,V1,32767);
  DBMS_OUTPUT.PUT_LINE('Get line: ' || V1);
  UTL_FILE.FCLOSE(F1);

  -- In this example, FOPEN's MAX_LINESIZE is NULL and defaults to 1024,
  -- so number of bytes returned will be 1024 or less if line terminator is seen.
  F1 := UTL_FILE.FOPEN('UTL_FILE_TEMP','u12345.tmp','R');
  UTL_FILE.GET_LINE(F1,V1,32767);
  DBMS_OUTPUT.PUT_LINE('Get line: ' || V1);
  UTL_FILE.FCLOSE(F1);

  -- GET_LINE doesn't specify a number of bytes, so it defaults to
  -- same value as FOPEN's MAX_LINESIZE which is NULL and defaults to 1024.
  -- So number of bytes returned will be 1024 or less if line terminator is seen.
  F1 := UTL_FILE.FOPEN('UTL_FILE_TEMP','u12345.tmp','R');
  UTL_FILE.GET_LINE(F1,V1);
  DBMS_OUTPUT.PUT_LINE('Get line: ' || V1);
  UTL_FILE.FCLOSE(F1);
END;
```

Consider the following test file, `u12345.tmp`, in the `utl_file_temp` directory:

```
This is line 1.
This is line 2.
This is line 3.
This is line 4.
This is line 5.
```

The example results in the following output, repeatedly getting the first line only:

```
Get line: This is line 1.
Get line: This is line 1.
Get line: This is line 1.

PL/SQL procedure successfully completed.
```

Example 2

This example appends content to the end of a file using the `PUTF` procedure.

```
declare
   handle utl_file.file_type;
   my_world  varchar2(4) := 'Zork';

begin
   handle := utl_file.fopen('UTL_FILE_TEMP','u12345.tmp','a');
   utl_file.putf(handle, '\nHello, world!\nI come from %s with %s.\n', my_world,
                      'greetings for all earthlings');
   utl_file.fflush(handle);
   utl_file.fclose(handle);
end;
/
```

This appends the following to file `u12345.tmp` in the `utl_file_temp` directory:

```
Hello, world!
I come from Zork with greetings for all earthlings.
```

Example 3

This procedure gets raw data from a specified file using the `GET_RAW` procedure. It exits when it reaches the end of the data, through its handling of `NO_DATA_FOUND` in the `EXCEPTION` processing.

```
CREATE OR REPLACE PROCEDURE getraw(n IN VARCHAR2) IS
  h     UTL_FILE.FILE_TYPE;
  Buf   RAW(32767);
  Amnt  CONSTANT BINARY_INTEGER := 32767;
BEGIN
  h := UTL_FILE.FOPEN('UTL_FILE_TEMP', n, 'r', 32767);
  LOOP
    BEGIN
      UTL_FILE.GET_RAW(h, Buf, Amnt);

      -- Do something with this chunk
      DBMS_OUTPUT.PUT_LINE('This is the raw data:');
      DBMS_OUTPUT.PUT_LINE(Buf);
    EXCEPTION WHEN No_Data_Found THEN
      EXIT;
    END;
  END LOOP;
  UTL_FILE.FCLOSE (h);
END;
/
```

Consider the following content in file `u12345.tmp` in the `utl_file_temp` directory:

```
hello world!
```

The example produces output as follows:

```
Command> begin
       > getraw('u12345.tmp');
       > end;
       > /
This is the raw data:
68656C6C6F20776F726C64210A

PL/SQL procedure successfully completed.
```

Data structures

The `UTL_FILE` package defines a record type.

Record types

- FILE_TYPE record type

FILE_TYPE record type

The contents of `FILE_TYPE` are private to the `UTL_FILE` package. You should not reference or change components of this record.

```
TYPE file_type IS RECORD (
   id          BINARY_INTEGER,
   datatype    BINARY_INTEGER,
   byte_mode   BOOLEAN);
```

Fields

Table 9–2 FILE_TYPE fields

Field	Description
id	A numeric value indicating the internal file handle number.
datatype	Indicates whether the file is a `CHAR` file, `NCHAR` file, or other (binary).
byte_mode	Indicates whether the file was open as a binary file or as a text file.

Important: Oracle does not guarantee the persistence of `FILE_TYPE` values between database sessions or within a single session. Attempts to clone file handles or use dummy file handles may have indeterminate outcomes.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

Summary of UTL_FILE subprograms

Table 9–3 UTL_FILE Subprograms

Subprogram	Description
FCLOSE procedure	Closes a file.
FCLOSE_ALL procedure	Closes all open file handles.
FCOPY procedure	Copies a contiguous portion of a file to a newly created file.
FFLUSH procedure	Physically writes all pending output to a file.
FGETATTR procedure	Reads and returns the attributes of a disk file.
FGETPOS function	Returns the current relative offset position (in bytes) within a file, in bytes.
FOPEN function	Opens a file for input or output.
FOPEN_NCHAR function	Opens a file in Unicode for input or output.
FREMOVE procedure	Deletes a disk file if you have sufficient privileges.
FRENAME procedure	Renames an existing file to a new name, similar to the UNIX `mv` function.
FSEEK procedure	Adjusts the file pointer forward or backward within the file by the number of bytes specified.
GET_LINE procedure	Reads text from an open file.
GET_LINE_NCHAR procedure	Reads text in Unicode from an open file.
GET_RAW procedure	Reads a `RAW` string value from a file and adjusts the file pointer ahead by the number of bytes read.
IS_OPEN function	Determines if a file handle refers to an open file.
NEW_LINE procedure	Writes one or more operating system-specific line terminators to a file.
PUT procedure	Writes a string to a file.
PUT_LINE procedure	Writes a line to a file, and so appends an operating system-specific line terminator.
PUT_LINE_NCHAR procedure	Writes a Unicode line to a file.
PUT_NCHAR procedure	Writes a Unicode string to a file.
PUT_RAW procedure	Accepts as input a `RAW` data value and writes the value to the output buffer.
PUTF procedure	Equivalent to `PUT` but with formatting.
PUTF_NCHAR procedure	Equivalent to `PUT_NCHAR` but with formatting.

FCLOSE procedure

This procedure closes an open file identified by a file handle.

Syntax

```
UTL_FILE.FCLOSE (
   file IN OUT UTL_FILE.FILE_TYPE);
```

Parameters

Table 9–4 FCLOSE procedure parameters

Parameter	Description
`file`	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
WRITE_ERROR
```

If there is buffered data yet to be written when `FCLOSE` runs, you may receive `WRITE_ERROR` when closing a file.

Examples

See "Examples" on page 9-7.

FCLOSE_ALL procedure

This procedure closes all open file handles for the session. This is useful as an emergency cleanup procedure, such as after a PL/SQL program exits on an exception.

Syntax

```
UTL_FILE.FCLOSE_ALL;
```

Usage notes

`FCLOSE_ALL` does not alter the state of the open file handles held by the user. Therefore, an `IS_OPEN` test on a file handle after an `FCLOSE_ALL` call still returns `TRUE`, even though the file has been closed. No further read or write operations can be performed on a file that was open before an `FCLOSE_ALL`.

Exceptions

Refer to "Exceptions" on page 9-6 for information about this exception.

```
WRITE_ERROR
```

FCOPY procedure

This procedure copies a contiguous portion of a file to a newly created file. By default, the whole file is copied if the `start_line` and `end_line` parameters are omitted. The source file is opened in read mode. The destination file is opened in write mode. A starting and ending line number can optionally be specified to select a portion from the center of the source file for copying.

Syntax

```
UTL_FILE.FCOPY (
   src_location    IN VARCHAR2,
   src_filename    IN VARCHAR2,
   dest_location   IN VARCHAR2,
   dest_filename   IN VARCHAR2,
  [start_line      IN BINARY_INTEGER DEFAULT 1,
   end_line        IN BINARY_INTEGER DEFAULT NULL]);
```

Parameters

Table 9–5   FCOPY procedure parameters

Parameters	Description
`src_location`	Directory location of the source file.
`src_filename`	Source file to be copied.
`dest_location`	Destination directory where the destination file is created.
`dest_filename`	Destination file created from the source file.
`start_line`	Line number at which to begin copying. The default is `1` for the first line.
`end_line`	Line number at which to stop copying. The default is `NULL`, signifying end of file.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILENAME
INVALID_PATH
INVALID_OPERATION
INVALID_OFFSET
READ_ERROR
WRITE_ERROR
```

FFLUSH procedure

`FFLUSH` physically writes pending data to the file identified by the file handle. Normally, data being written to a file is buffered. The `FFLUSH` procedure forces the buffered data to be written to the file. The data must be terminated with a newline character.

Flushing is useful when the file must be read while still open. For example, debugging messages can be flushed to the file so that they can be read immediately.

Syntax

```
UTL_FILE.FFLUSH (
   file  IN UTL_FILE.FILE_TYPE);
```

Parameters

Table 9–6 FFLUSH procedure parameters

Parameters	**Description**
`file`	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
```

Examples

See "Examples" on page 9-7.

FGETATTR procedure

This procedure reads and returns the attributes of a disk file.

Syntax

```
UTL_FILE.FGETATTR(
   location    IN VARCHAR2,
   filename    IN VARCHAR2,
   fexists     OUT BOOLEAN,
   file_length OUT NUMBER,
   block_size   OUT BINARY_INTEGER);
```

Parameters

Table 9–7 FGETATTR procedure parameters

Parameters	Description
`location`	Location of the source file.
`filename`	The name of the file to be examined.
`fexists`	A `BOOLEAN` for whether the file exists.
`file_length`	The length of the file in bytes. `NULL` if file does not exist.
`block_size`	The file system block size in bytes. `NULL` if the file does not exist.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_PATH
INVALID_FILENAME
INVALID_OPERATION
READ_ERROR
ACCESS_DENIED
```

FGETPOS function

This function returns the current relative offset position within a file, in bytes.

Syntax

```
UTL_FILE.FGETPOS (
   file   IN utl_file.file_type)
 RETURN BINARY_INTEGER;
```

Parameters

Table 9–8 FGETPOS function parameters

Parameters	Description
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.

Return value

The relative offset position for an open file, in bytes. It returns 0 for the beginning of the file.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
```

An `INVALID_FILEHANDLE` exception is raised if the file is not open. An `INVALID_OPERATION` exception is raised if the file was opened for byte mode operations.

FOPEN function

This function opens a file. You can specify the maximum line size and have a maximum of 50 files open simultaneously. Also see "FOPEN_NCHAR function" on page 9-20.

Syntax

```
UTL_FILE.FOPEN (
   location     IN VARCHAR2,
   filename     IN VARCHAR2,
   open_mode    IN VARCHAR2,
   max_linesize IN BINARY_INTEGER DEFAULT 1024)
  RETURN utl_file.file_type;
```

Parameters

Table 9–9   FOPEN function parameters

Parameter	Description
`location`	Directory location of file.
`filename`	File name, including extension (file type), without directory path. If a directory path is given as a part of the file name, it is ignored by `FOPEN`. On UNIX, the file name cannot end with a slash, "`/`".
`open_mode`	Specifies how the file is opened. Modes include: ■ `r` - read text ■ `w` - write text ■ `a` - append text ■ `rb` - read byte mode ■ `wb` - write byte mode ■ `ab` - append byte mode If you try to open a file specifying '`a`' or '`ab`' for *`open_mode`* but the file does not exist, the file is created in `WRITE` mode.
`max_linesize`	Maximum number of characters for each line, including the newline character, for this file (minimum value 1, maximum value 32767). If unspecified, TimesTen supplies a default value of 1024.

Return value

A file handle, which must be passed to all subsequent procedures that operate on that file. The specific contents of the file handle are private to the `UTL_FILE` package, and individual components should not be referenced or changed by the `UTL_FILE` user.

Usage notes

The file location and file name parameters are supplied to the `FOPEN` function as separate strings, so that the file location can be checked against the `utl_file_temp` directory. Together, the file location and name must represent a legal file name on the system, and the directory must be accessible. Any subdirectories of `utl_file_temp` are not accessible.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_PATH
INVALID_MODE
INVALID_OPERATION
INVALID_MAXLINESIZE
```

Examples

See "Examples" on page 9-7.

FOPEN_NCHAR function

This function opens a file in national character set mode for input or output, with the maximum line size specified. You can have a maximum of 50 files open simultaneously. With this function, you can read or write a text file in Unicode instead of in the database character set.

Even though the contents of an `NVARCHAR2` buffer may be `AL16UTF16` or UTF-8 (depending on the national character set of the database), the contents of the file are always read and written in UTF-8. `UTL_FILE` converts between UTF-8 and `AL16UTF16` as necessary.

Also see "FOPEN function" on page 9-18.

Syntax

```
UTL_FILE.FOPEN_NCHAR (
   location     IN VARCHAR2,
   filename     IN VARCHAR2,
   open_mode    IN VARCHAR2,
   max_linesize IN BINARY_INTEGER DEFAULT 1024)
RETURN utl_file.file_type;
```

Parameters

Table 9–10 FOPEN_NCHAR function parameters

Parameter	Description
location	Directory location of file.
filename	File name, including extension.
open_mode	Open mode: `r`, `w`, `a`, `rb`, `wb`, or `ab` (as documented for `FOPEN`).
max_linesize	Maximum number of characters for each line, including the newline character, for this file. The minimum value is 1. The maximum is 32767.

Return value

A file handle, which must be passed to all subsequent procedures that operate on that file. The specific contents of the file handle are private to the `UTL_FILE` package, and individual components should not be referenced or changed by the `UTL_FILE` user.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_PATH
INVALID_MODE
INVALID_OPERATION
INVALID_MAXLINESIZE
```

FREMOVE procedure

This procedure deletes a disk file if you have sufficient privileges.

Syntax

```
UTL_FILE.FREMOVE (
   location IN VARCHAR2,
   filename IN VARCHAR2);
```

Parameters

Table 9–11 FREMOVE procedure parameters

Parameters	Description
location	The directory location of the file.
filename	The name of the file to be deleted.

Usage notes

This procedure does not verify privileges before deleting a file. The operating system verifies file and directory permissions.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_PATH
INVALID_FILENAME
INVALID_OPERATION
ACCESS_DENIED
DELETE_FAILED
```

FRENAME procedure

This procedure renames an existing file.

Syntax

```
UTL_FILE.FRENAME (
   src_location  IN VARCHAR2,
   src_filename  IN VARCHAR2,
   dest_location IN VARCHAR2,
   dest_filename IN VARCHAR2,
   overwrite     IN BOOLEAN DEFAULT FALSE);
```

Parameters

Table 9–12 FRENAME procedure parameters

Parameters	Description
src_location	The directory location of the source file.
src_filename	The source file to be renamed.
dest_location	The destination directory of the destination file.
dest_filename	The new name of the file.
overwrite	Whether it is permissible to overwrite an existing file in the destination directory. The default is `FALSE`.

Usage notes

Permission on both the source and destination directories must be granted.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_PATH
INVALID_FILENAME
RENAME_FAILED
ACCESS_DENIED
```

FSEEK procedure

This procedure adjusts the file pointer forward or backward within the file by the number of bytes specified.

Syntax

```
UTL_FILE.FSEEK (
   file            IN OUT utl_file.file_type,
   absolute_offset IN     BINARY_INTEGER DEFAULT NULL,
   relative_offset IN     BINARY_INTEGER DEFAULT NULL);
```

Parameters

Table 9–13 FSEEK procedure parameters

Parameters	**Description**
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.
absolute_offset	The absolute location to which to seek, in bytes. Default is `NULL`.
relative_offset	The number of bytes to seek forward or backward. Use a positive integer to seek forward, a negative integer to see backward, or 0 for the current position. Default is `NULL`.

Usage notes

- Using `FSEEK`, you can read previous lines in the file without first closing and reopening the file. You must know the number of bytes by which you want to navigate.
- If the beginning of the file is reached before the number of bytes specified, then the file pointer is placed at the beginning of the file.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
INVALID_OFFSET
```

`INVALID_OPERATION` is raised if the file was opened for byte-mode operations. `INVALID_OFFSET` is raised if the end of the file is reached before the number of bytes specified.

GET_LINE procedure

This procedure reads text from the open file identified by the file handle and places the text in the output buffer parameter. Text is read up to, but not including, the line terminator, or up to the end of the file, or up to the end of the `len` parameter. It cannot exceed the *`max_linesize`* specified in `FOPEN`.

Syntax

```
UTL_FILE.GET_LINE (
   file        IN  UTL_FILE.FILE_TYPE,
   buffer      OUT VARCHAR2,
   len         IN  BINARY_INTEGER DEFAULT NULL);
```

Parameters

Table 9–14 GET_LINE procedure parameters

Parameters	Description
`file`	Active file handle returned by an `FOPEN` call.
`buffer`	Data buffer to receive the line read from the file.
`len`	The number of bytes read from the file. Default is `NULL`. If `NULL`, TimesTen supplies the value of *`max_linesize`* from `FOPEN`.

Usage notes

- Because the line terminator character is not read into the buffer, reading blank lines returns empty strings.
- The maximum size of the *`buffer`* parameter is 32767 bytes unless you specify a smaller size in `FOPEN`.
- If unspecified, TimesTen supplies a default value of 1024. Also see "GET_LINE_NCHAR procedure" on page 9-25.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
CHARSETMISMATCH
NO_DATA_FOUND
VALUE_ERROR
```

`INVALID_OPERATION` is thrown if the file was not opened for read mode (mode `r`) or was opened for byte-mode operations. `CHARSETMISMATCH` is thrown if `FOPEN_NCHAR` was used instead of `FOPEN` to open the file. `NO_DATA_FOUND` is thrown if no text was read due to end of file. `VALUE_ERROR` is thrown if the line does not fit into the buffer. (`NO_DATA_FOUND` and `VALUE_ERROR` are predefined PL/SQL exceptions.)

Examples

See "Examples" on page 9-7.

GET_LINE_NCHAR procedure

This procedure reads text from the open file identified by the file handle and places the text in the output buffer parameter. With this function, you can read a text file in Unicode instead of in the database character set.

The file must be opened in national character set mode, and must be encoded in the UTF-8 character set. The expected buffer data type is `NVARCHAR2`. If a variable of another data type such as `NCHAR` or `VARCHAR2` is specified, PL/SQL will perform standard implicit conversion from `NVARCHAR2` after the text is read.

Also see "GET_LINE procedure" on page 9-24.

Syntax

```
UTL_FILE.GET_LINE_NCHAR (
   file        IN  UTL_FILE.FILE_TYPE,
   buffer      OUT NVARCHAR2,
   len         IN  BINARY_INTEGER DEFAULT NULL);
```

Parameters

Table 9–15 GET_LINE_NCHAR procedure parameters

Parameters	Description
`file`	Active file handle returned by an `FOPEN_NCHAR` call. The file must be open for reading (mode `r`).
`buffer`	Data buffer to receive the line read from the file.
`len`	The number of bytes read from the file. Default is `NULL`. If `NULL`, TimesTen supplies the value of *`max_linesize`* from `FOPEN_NCHAR`.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
CHARSETMISMATCH
NO_DATA_FOUND
VALUE_ERROR
```

`INVALID_OPERATION` is thrown if the file was not opened for read mode (mode `r`) or was opened for byte-mode operations. `NO_DATA_FOUND` is thrown if no text was read due to end of file. `VALUE_ERROR` is thrown if the line does not fit into the buffer. `CHARSETMISMATCH` is thrown if the file was opened by `FOPEN` instead of `FOPEN_NCHAR`. (`NO_DATA_FOUND` and `VALUE_ERROR` are predefined PL/SQL exceptions.)

GET_RAW procedure

This procedure reads a `RAW` string value from a file and adjusts the file pointer ahead by the number of bytes read. It ignores line terminators.

Syntax

```
UTL_FILE.GET_RAW (
   file    IN  utl_file.file_type,
   buffer  OUT NOCOPY RAW,
   len     IN  BINARY_INTEGER DEFAULT NULL);
```

Parameters

Table 9–16 GET_RAW function parameters

Parameters	Description
`file`	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.
`buffer`	The `RAW` data.
`len`	The number of bytes read from the file. Default is `NULL`. If `NULL`, *`len`* is assumed to be the maximum length of `RAW`.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
READ_ERROR
LENGTH_MISMATCH
NO_DATA_FOUND
```

(`NO_DATA_FOUND` is a predefined PL/SQL exception.)

Examples

See "Examples" on page 9-7.

IS_OPEN function

This function tests a file handle to see if it identifies an open file. `IS_OPEN` reports only whether a file handle represents a file that has been opened, but not yet closed. It does not guarantee that there will be no operating system errors when you attempt to use the file handle.

Syntax

```
UTL_FILE.IS_OPEN (
   file  IN UTL_FILE.FILE_TYPE)
  RETURN BOOLEAN;
```

Parameters

Table 9–17 IS_OPEN function parameters

Parameter	Description
`file`	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.

Return value

`TRUE` if the file is open, or `FALSE` if not.

Exceptions

Refer to "Exceptions" on page 9-6 for information about this exception.

`INVALID_FILEHANDLE`

NEW_LINE procedure

This procedure writes one or more line terminators to the file identified by the input file handle. This procedure is distinct from `PUT` because the line terminator is a platform-specific character or sequence of characters.

Syntax

```
UTL_FILE.NEW_LINE (
   file     IN UTL_FILE.FILE_TYPE,
   lines    IN BINARY_INTEGER := 1);
```

Parameters

Table 9–18 NEW_LINE procedure parameters

Parameters	Description
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.
lines	Number of line terminators to be written to the file.

Exceptions

Refer to "Exceptions" on page 9-6 for information about this exception.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
```

PUT procedure

`PUT` writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. No line terminator is appended by `PUT`. Use `NEW_LINE` to terminate the line or `PUT_LINE` to write a complete line with a line terminator. Also see "PUT_NCHAR procedure" on page 9-32.

Syntax

```
UTL_FILE.PUT (
   file      IN UTL_FILE.FILE_TYPE,
   buffer    IN VARCHAR2);
```

Parameters

Table 9–19 PUT procedure parameters

Parameters	Description
file	Active file handle returned by an `FOPEN_NCHAR` call. The file must be open for writing.
buffer	Buffer that contains the text to be written to the file.

Usage notes

The maximum size of the *`buffer`* parameter is 32767 bytes unless you specify a smaller size in `FOPEN`. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential `PUT` calls cannot exceed 32767 without intermediate buffer flushes.

Exceptions

Refer to "Exceptions" on page 9-6 for information about this exception.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was not opened using mode `w` or `a` (write or append). `CHARSETMISMATCH` is thrown if `FOPEN_NCHAR` was used instead of `FOPEN` to open the file.

PUT_LINE procedure

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. The file must be open for write operations. `PUT_LINE` terminates the line with the platform-specific line terminator character or characters. Also see "PUT_LINE_NCHAR procedure" on page 9-31.

Syntax

```
UTL_FILE.PUT_LINE (
   file      IN UTL_FILE.FILE_TYPE,
   buffer    IN VARCHAR2,
   autoflush IN BOOLEAN DEFAULT FALSE);
```

Parameters

Table 9–20 PUT_LINE procedure parameters

Parameters	Description
`file`	Active file handle returned by an `FOPEN` call.
`buffer`	Text buffer that contains the lines to be written to the file.
`autoflush`	Flushes the buffer to disk after the write.

Usage notes

The maximum size of the `buffer` parameter is 32767 bytes unless you specify a smaller size in `FOPEN`. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential `PUT` calls cannot exceed 32767 without intermediate buffer flushes.

Exceptions

Refer to "Exceptions" on page 9-6 for information about this exception.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was opened for byte-mode operations. `CHARSETMISMATCH` is thrown if `FOPEN_NCHAR` was used instead of `FOPEN` to open the file.

PUT_LINE_NCHAR procedure

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle. With this function, you can write a text file in Unicode instead of in the database character set. This procedure is equivalent to the PUT_NCHAR procedure, except that the line separator is appended to the written text. Also see "PUT_LINE procedure" on page 9-30.

Syntax

```
UTL_FILE.PUT_LINE_NCHAR (
   file    IN UTL_FILE.FILE_TYPE,
   buffer  IN NVARCHAR2);
```

Parameters

Table 9–21 PUT_LINE_NCHAR procedure parameters

Parameters	Description
file	Active file handle returned by an `FOPEN_NCHAR` call. The file must be open for writing.
buffer	Text buffer that contains the lines to be written to the file.

Usage notes

The maximum size of the *buffer* parameter is 32767 bytes unless you specify a smaller size in `FOPEN`. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential `PUT` calls cannot exceed 32767 without intermediate buffer flushes.

Exceptions

Refer to "Exceptions" on page 9-6 for information about this exception.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was opened for byte-mode operations. `CHARSETMISMATCH` is thrown if `FOPEN` was used instead of `FOPEN_NCHAR` to open the file.

PUT_NCHAR procedure

This procedure writes the text string stored in the buffer parameter to the open file identified by the file handle.

With this function, you can write a text file in Unicode instead of in the database character set. The file must be opened in the national character set mode. The text string will be written in the UTF-8 character set. The expected buffer data type is `NVARCHAR2`. If a variable of another data type is specified, PL/SQL will perform implicit conversion to `NVARCHAR2` before writing the text.

Also see "PUT procedure" on page 9-29.

Syntax

```
UTL_FILE.PUT_NCHAR (
   file      IN UTL_FILE.FILE_TYPE,
   buffer    IN NVARCHAR2);
```

Parameters

Table 9–22 PUT_NCHAR procedure parameters

Parameters	Description
file	Active file handle returned by an `FOPEN_NCHAR` call.
buffer	Buffer that contains the text to be written to the file.

Usage notes

The maximum size of the *`buffer`* parameter is 32767 bytes unless you specify a smaller size in `FOPEN`. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential `PUT` calls cannot exceed 32767 without intermediate buffer flushes.

Exceptions

Refer to "Exceptions" on page 9-6 for information about this exception.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was not opened using mode `w` or `a` (write or append). `CHARSETMISMATCH` is thrown if the file was opened by `FOPEN` instead of `FOPEN_NCHAR`.

PUT_RAW procedure

This procedure accepts as input a `RAW` data value and writes the value to the output buffer.

Syntax

```
UTL_FILE.PUT_RAW (
   file       IN utl_file.file_type,
   buffer     IN RAW,
   autoflush  IN BOOLEAN DEFAULT FALSE);
```

Parameters

Table 9–23   PUT_RAW procedure parameters

Parameters	**Description**
file	Active file handle returned by an `FOPEN` or `FOPEN_NCHAR` call.
buffer	The `RAW` data written to the buffer.
autoflush	If `TRUE`, performs a flush after writing the value to the output buffer. The default is `FALSE`.

Usage notes

You can request an automatic flush of the buffer by setting *autoflush* to `TRUE`.

The maximum size of the *buffer* parameter is 32767 bytes unless you specify a smaller size in `FOPEN`. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential `PUT` calls cannot exceed 32767 without intermediate buffer flushes.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
```

PUTF procedure

This procedure is a formatted `PUT` procedure. It works like a limited `printf()`. Also see "PUTF_NCHAR procedure" on page 9-35.

Syntax

```
UTL_FILE.PUTF (
   file    IN UTL_FILE.FILE_TYPE,
   format  IN VARCHAR2,
   [arg1   IN VARCHAR2  DEFAULT NULL,
   . . .
   arg5    IN VARCHAR2  DEFAULT NULL]);
```

Parameters

Table 9–24 PUTF procedure parameters

Parameters	Description
file	Active file handle returned by an `FOPEN` call.
format	Format string that can contain text and the formatting characters `\n` and `%s`.
arg1..arg5	From one to five operational argument strings. Argument strings are substituted, in order, for the `%s` formatters in the format string. If there are more formatters in the format parameter string than there are arguments, an empty string is substituted for each `%s` for which there is no argument.

Usage notes

The format string can contain any text, but the character sequences `%s` and `\n` have special meaning.

Character sequence	Meaning
`%s`	Substitute this sequence with the string value of the next argument in the argument list.
`\n`	Substitute with the appropriate platform-specific line terminator.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was opened for byte-mode operations. `CHARSETMISMATCH` is thrown if `FOPEN_NCHAR` was used instead of `FOPEN` to open the file.

Examples

See "Examples" on page 9-7.

PUTF_NCHAR procedure

This procedure is the formatted version of the PUT_NCHAR procedure. Using `PUTF_NCHAR`, you can write a text file in Unicode instead of in the database character set. It accepts a format string with formatting elements `\n` and `%s`, and up to five arguments to be substituted for consecutive occurrences of `%s` in the format string. The expected data type of the format string and the arguments is `NVARCHAR2`.

If variables of another data type are specified, PL/SQL will perform implicit conversion to `NVARCHAR2` before formatting the text. Formatted text is written in the UTF-8 character set to the file identified by the file handle. The file must be opened in the national character set mode.

Syntax

```
UTL_FILE.PUTF_NCHAR (
   file    IN UTL_FILE.FILE_TYPE,
   format  IN NVARCHAR2,
   [arg1   IN NVARCHAR2  DEFAULT NULL,
   . . .
   arg5    IN NVARCHAR2  DEFAULT NULL]);
```

Parameters

Table 9–25 PUTF_NCHAR procedure parameters

Parameters	Description
`file`	Active file handle returned by an `FOPEN_NCHAR` call. The file must be open for reading (mode `r`).
`format`	Format string that can contain text and the format characters `\n` and `%s`.
`arg1..arg5`	From one to five operational argument strings. Argument strings are substituted, in order, for the `%s` format characters in the format string. If there are more format characters in the format string than there are arguments, an empty string is substituted for each `%s` for which there is no argument.

Usage notes

The maximum size of the *`buffer`* parameter is 32767 bytes unless you specify a smaller size in `FOPEN`. If unspecified, TimesTen supplies a default value of 1024. The sum of all sequential `PUT` calls cannot exceed 32767 without intermediate buffer flushes.

Exceptions

Refer to "Exceptions" on page 9-6 for information about these exceptions.

```
INVALID_FILEHANDLE
INVALID_OPERATION
WRITE_ERROR
CHARSETMISMATCH
```

`INVALID_OPERATION` is thrown if the file was opened for byte-mode operations. `CHARSETMISMATCH` is thrown if the file was opened by `FOPEN` instead of `FOPEN_NCHAR`.

10

UTL_IDENT

The `UTL_IDENT` package indicates which database or client PL/SQL is running in, such as TimesTen versus Oracle Database, and server versus client. Each database or client running PL/SQL has its own copy of this package.

This chapter contains the following topics:

- Using UTL_IDENT
  - Overview
  - Security model
  - Constants
  - Examples

Using UTL_IDENT

This section contains topics that relate to using the `UTL_IDENT` package.

- Overview
- Security model
- Constants
- Examples

Overview

The `UTL_IDENT` package indicates whether PL/SQL is running on TimesTen, an Oracle Database client, an Oracle Database server, or Oracle Forms. Each of these has its own version of `UTL_IDENT` with appropriate settings for the constants.

The primary use case for the `UTL_IDENT` package is for conditional compilation, resembling the following, of PL/SQL packages that are supported by Oracle Database, TimesTen, or clients such as Oracle Forms.

```
$if utl_ident.is_oracle_server $then
    [...Run code supported for Oracle Database...]
$elsif utl_ident.is_timesten $then
    [...code supported for TimesTen Database...]
$end
```

Also see "Examples" on page 10-6.

Security model

The `UTL_IDENT` package runs as the package owner `SYS`. The public synonym `UTL_IDENT` and `EXECUTE` permission on this package are granted to `PUBLIC`.

Constants

The `UTL_IDENT` package uses the constants shown in Table 10–1, which indicates the settings for TimesTen.

Table 10–1 UTL_IDENT constants

Constant	Type	Value	Description
`IS_ORACLE_SERVER`	`BOOLEAN`	`FALSE`	Stipulates whether Oracle Database.
`IS_ORACLE_CLIENT`	`BOOLEAN`	`FALSE`	Stipulates whether Oracle Client.
`IS_ORACLE_FORMS`	`BOOLEAN`	`FALSE`	Stipulates whether Oracle Forms.
`IS_TIMESTEN`	`BOOLEAN`	`TRUE`	Stipulates whether TimesTen.

Examples

This example uses the `UTL_IDENT` and `TT_DB_VERSION` packages to show information about the database being used. For the current release, it displays either "Oracle Database 11.2" or "TimesTen 11.2.1". The conditional compilation trigger character, `$`, identifies code that is processed before the application is compiled.

```
Command> run what_db.sql

create or replace function what_db
return varchar2
as
 dbname varchar2(100);
 version varchar2(100);
begin
$if utl_ident.is_timesten
$then
 dbname := 'TimesTen';
 version := substr(tt_db_version.version, 1, 2) ||
            '.' ||
            substr(tt_db_version.version, 3, 1) ||
            '.' ||
            substr(tt_db_version.version, 4, 1);
$elsif utl_ident.is_oracle_server
$then
 dbname := 'Oracle Database';
 version := dbms_db_version.version || '.' || dbms_db_version.release;
$else
 dbname := 'Non-database environment';
 version := '';
$end
 return dbname || ' ' || version;
end;
/

Function created.

set serveroutput on;

begin
dbms_output.put_line(what_db());
end;
/

TimesTen 11.2.1

PL/SQL procedure successfully completed.
```

11

UTL_RAW

The `UTL_RAW` package provides SQL functions for manipulating `RAW` data types.

This chapter contains the following topics:

- Using UTL_RAW
  - Overview
  - Operational notes
- Summary of UTL_RAW subprograms

Using UTL_RAW

- Overview
- Operational notes

Overview

This package is necessary because normal SQL functions do not operate on `RAW` values and PL/SQL does not allow overloading between a `RAW` and a `CHAR` data type.

`UTL_RAW` is not specific to the database environment and may be used in other environments. For this reason, the prefix `UTL` has been given to the package, instead of `DBMS`.

Operational notes

`UTL_RAW` allows a `RAW` record to be composed of many elements. By using the `RAW` data type, character set conversion will not be performed, keeping the `RAW` value in its original format when being transferred through remote procedure calls.

With the `RAW` functions, you can manipulate binary data that was previously limited to the `hextoraw` and `rawtohex` SQL functions.

Functions returning `RAW` values do so in hexadecimal encoding.

Summary of UTL_RAW subprograms

Table 11–1 UTL_RAW Package Subprograms

Subprogram	Description
BIT_AND function	Performs bitwise logical `AND` of two `RAW` values and returns the resulting `RAW`.
BIT_COMPLEMENT function	Performs bitwise logical `COMPLEMENT` of a `RAW` value and returns the resulting `RAW`.
BIT_OR function	Performs bitwise logical `OR` of two `RAW` values and returns the resulting `RAW`.
BIT_XOR function	Performs bitwise logical `XOR` ("exclusive or") of two `RAW` values and returns the resulting `RAW`.
CAST_FROM_BINARY_DOUBLE function	Returns the `RAW` binary representation of a `BINARY_DOUBLE` value.
CAST_FROM_BINARY_FLOAT function	Returns the `RAW` binary representation of a `BINARY_FLOAT` value.
CAST_FROM_BINARY_INTEGER function	Returns the `RAW` binary representation of a `BINARY_INTEGER` value.
CAST_FROM_NUMBER function	Returns the `RAW` binary representation of a `NUMBER` value.
CAST_TO_BINARY_DOUBLE function	Casts the `RAW` binary representation of a `BINARY_DOUBLE` value into a `BINARY_DOUBLE`.
CAST_TO_BINARY_FLOAT function	Casts the `RAW` binary representation of a `BINARY_FLOAT` value into a `BINARY_FLOAT`.
CAST_TO_BINARY_INTEGER function	Casts the `RAW` binary representation of a `BINARY_INTEGER` value into a `BINARY_INTEGER`.
CAST_TO_NUMBER function	Casts the `RAW` binary representation of a `NUMBER` value into a `NUMBER`.
CAST_TO_NVARCHAR2 function	Converts a `RAW` value into an `NVARCHAR2` value.
CAST_TO_RAW function	Converts a `VARCHAR2` value into a `RAW` value.
CAST_TO_VARCHAR2 function	Converts a `RAW` value into a `VARCHAR2` value.
COMPARE function	Compares two `RAW` values.
CONCAT function	Concatenates up to 12 `RAW` values into a single `RAW`.
CONVERT function	Converts a `RAW` value from one character set to another and returns the resulting `RAW`.
COPIES function	Copies a `RAW` value a specified number of times and returns the concatenated `RAW` value.
LENGTH function	Returns the length in bytes of a `RAW` value.
OVERLAY function	Overlays the specified portion of a target `RAW` value with an overlay `RAW` value, starting from a specified byte position and proceeding for a specified number of bytes.
REVERSE function	Reverses a byte-sequence in a `RAW` value.
SUBSTR function	Returns a substring of a `RAW` value for a specified number of bytes from a specified starting position.
TRANSLATE function	Translates the specified bytes from an input `RAW` value according to the bytes in a specified translation `RAW` value.

Table 11–1 (Cont.) UTL_RAW Package Subprograms

Subprogram	Description
TRANSLITERATE function	Converts the specified bytes from an input `RAW` value according to the bytes in a specified transliteration `RAW` value.
XRANGE function	Returns a `RAW` value containing the succession of one-byte encodings beginning and ending with the specified byte-codes.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

BIT_AND function

This function performs bitwise logical `AND` of two supplied `RAW` values and returns the resulting `RAW`.

Syntax

```
UTL_RAW.BIT_AND (
   r1 IN RAW,
   r2 IN RAW)
RETURN RAW;
```

Parameters

Table 11–2   BIT_AND function parameters

Parameter	Description
`r1`	First `RAW` value for `AND` operation.
`r2`	Second `RAW` value for `AND` operation.

Return value

Contains the result of the `AND` operation, or `NULL` if either input value is `NULL`.

Usage notes

If *`r1`* and *`r2`* differ in length, the operation is terminated after the last byte of the shorter of the two `RAW` values, and the unprocessed portion of the longer `RAW` value is appended to the partial result. The resulting length equals that of the longer of the two input values.

BIT_COMPLEMENT function

This function performs bitwise logical `COMPLEMENT` of the supplied `RAW` value and returns the resulting `RAW`. The result length equals the input `RAW` length.

Syntax

```
UTL_RAW.BIT_COMPLEMENT (
   r IN RAW)
  RETURN RAW;
```

Parameters

Table 11–3 BIT_COMPLEMENT function parameters

Parameter	Description
r	`RAW` value for `COMPLEMENT` operation.

Return value

Contains the result of the `COMPLEMENT` operation, or `NULL` if the input value is `NULL`.

BIT_OR function

This function performs bitwise logical `OR` of two supplied `RAW` values and returns the resulting `RAW`.

Syntax

```
UTL_RAW.BIT_OR (
   r1 IN RAW,
   r2 IN RAW)
  RETURN RAW;
```

Parameters

Table 11–4 BIT_OR function parameters

Parameters	Description
r1	First `RAW` value for `OR` operation.
r2	Second `RAW` value for `OR` operation.

Return value

Contains the result of the `OR` operation, or `NULL` if either input value is `NULL`.

Usage notes

If *r1* and *r2* differ in length, the operation is terminated after the last byte of the shorter of the two `RAW` values, and the unprocessed portion of the longer `RAW` value is appended to the partial result. The resulting length equals that of the longer of the two input values.

BIT_XOR function

This function performs bitwise logical `XOR` ("exclusive or") of two supplied `RAW` values and returns the resulting `RAW`.

Syntax

```
UTL_RAW.BIT_XOR (
   r1 IN RAW,
   r2 IN RAW)
  RETURN RAW;
```

Parameters

Table 11–5 BIT_XOR function parameters

Parameter	Description
r1	First `RAW` value for `XOR` operation.
r2	Second `RAW` value for `XOR` operation.

Return value

Contains the result of the `XOR` operation, or `NULL` if either input value is `NULL`.

Usage notes

If *r1* and *r2* differ in length, the operation is terminated after the last byte of the shorter of the two `RAW` values, and the unprocessed portion of the longer `RAW` value is appended to the partial result. The resulting length equals that of the longer of the two input values.

CAST_FROM_BINARY_DOUBLE function

This function returns the `RAW` binary representation of a `BINARY_DOUBLE` value.

Syntax

```
UTL_RAW.CAST_FROM_BINARY_DOUBLE(
   n          IN BINARY_DOUBLE,
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Parameters

Table 11–6 CAST_FROM_BINARY_DOUBLE function parameters

Parameter	Description
`n`	The `BINARY_DOUBLE` value.
`endianess`	A `BINARY_INTEGER` value indicating the endianess. The function recognizes the defined constants `big_endian`, `little_endian`, and `machine_endian`. The default is `big_endian`.

Return value

The `RAW` binary representation of the `BINARY_DOUBLE` value, or `NULL` if the input is `NULL`.

Usage notes

- An eight-byte `BINARY_DOUBLE` value maps to the IEEE 754 double-precision format as follows:

```
byte 0: bit 63 ~ bit 56
byte 1: bit 55 ~ bit 48
byte 2: bit 47 ~ bit 40
byte 3: bit 39 ~ bit 32
byte 4: bit 31 ~ bit 24
byte 5: bit 23 ~ bit 16
byte 6: bit 15 ~ bit  8
byte 7: bit  7 ~ bit  0
```

- Parameter *`endianess`* specifies how the bytes of the `BINARY_DOUBLE` value are mapped to the bytes of the `RAW` value. In the following matrix, `rb0` to `rb7` refer to the bytes of the `RAW` and `db0` to `db7` refer to the bytes of the `BINARY_DOUBLE`.

	rb0	**rb1**	**rb2**	**rb3**	**rb4**	**rb5**	**rb6**	**rb7**
big_endian	db0	db1	db2	db3	db4	db5	db6	db7
little_endian	db7	db6	db5	db4	db3	db2	db1	db0

- When `machine_endian` is specified, the eight bytes of the `BINARY_DOUBLE` argument are copied straight across into the `RAW` return value. The effect is the same as if the user specified `big_endian` on a big-endian system or `little_endian` on a little-endian system.

CAST_FROM_BINARY_FLOAT function

This function returns the `RAW` binary representation of a `BINARY_FLOAT` value.

Syntax

```
UTL_RAW.CAST_FROM_BINARY_FLOAT(
   n          IN BINARY_FLOAT,
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Parameters

Table 11–7 CAST_FROM_BINARY_FLOAT function parameters

Parameter	**Description**
`n`	The `BINARY_FLOAT` value.
`endianess`	A `BINARY_INTEGER` value indicating the endianess. The function recognizes the defined constants `big_endian`, `little_endian`, and `machine_endian`. The default is `big_endian`.

Return value

The `RAW` binary representation of the `BINARY_FLOAT` value, or `NULL` if the input is `NULL`.

Usage notes

- A four-byte `BINARY_FLOAT` value maps to the IEEE 754 single-precision format as follows:

```
byte 0: bit 31 ~ bit 24
byte 1: bit 23 ~ bit 16
byte 2: bit 15 ~ bit  8
byte 3: bit 7 ~  bit  0
```

- The parameter *`endianess`* specifies how the bytes of the `BINARY_FLOAT` value are mapped to the bytes of the `RAW` value. In the following matrix, `rb0` to `rb3` refer to the bytes of the `RAW` and `fb0` to `fb3` refer to the bytes of the `BINARY_FLOAT`.

	rb0	**rb1**	**rb2**	**rb3**
big_endian	`fbo`	`fb1`	`fb2`	`fb3`
little_endian	`fb3`	`fb2`	`fb1`	`fb0`

- When `machine_endian` is specified, the four bytes of the `BINARY_FLOAT` argument are copied straight across into the `RAW` return value. The effect is the same as if the user specified `big_endian` on a big-endian system or `little_endian` on a little-endian system.

CAST_FROM_BINARY_INTEGER function

This function returns the `RAW` binary representation of a `BINARY_INTEGER` value.

Syntax

```
UTL_RAW.CAST_FROM_BINARY_INTEGER (
   n          IN BINARY_INTEGER
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN RAW;
```

Parameters

Table 11–8 CAST_FROM_BINARY_INTEGER function parameters

Parameter	Description
n	The `BINARY_INTEGER` value.
endianess	A `BINARY_INTEGER` value indicating the endianess. The function recognizes the defined constants `big_endian`, `little_endian`, and `machine_endian`. The default is `big_endian`.

Return value

The `RAW` binary representation of the `BINARY_INTEGER` value, or `NULL` if the input is `NULL`.

CAST_FROM_NUMBER function

This function returns the `RAW` binary representation of a `NUMBER` value.

Syntax

```
UTL_RAW.CAST_FROM_NUMBER (
   n  IN NUMBER)
 RETURN RAW;
```

Parameters

Table 11–9   CAST_FROM_NUMBER function parameters

Parameter	Description
n	The `NUMBER` value.

Return value

The `RAW` binary representation of the `NUMBER` value, or `NULL` if the input is `NULL`.

CAST_TO_BINARY_DOUBLE function

This function casts the `RAW` binary representation of a `BINARY_DOUBLE` value into a `BINARY_DOUBLE` value.

Syntax

```
UTL_RAW.CAST_TO_BINARY_DOUBLE (
   r          IN RAW
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN BINARY_DOUBLE;
```

Parameters

Table 11–10 CAST_TO_BINARY_DOUBLE function parameters

Parameter	Description
`r`	The `RAW` binary representation of a `BINARY_DOUBLE` value
`endianess`	A `BINARY_INTEGER` value indicating the endianess. The function recognizes the defined constants `big_endian`, `little_endian`, and `machine_endian`. The default is `big_endian`.

Return value

The `BINARY_DOUBLE` value, or `NULL` if the input is `NULL`.

Usage notes

- If the `RAW` argument is more than eight bytes, only the first eight bytes are used and the rest of the bytes are ignored. If the result is -0, +0 is returned. If the result is `NaN`, the value `BINARY_DOUBLE_NAN` is returned.
- An eight-byte `BINARY_DOUBLE` value maps to the IEEE 754 double-precision format as follows:

```
byte 0: bit 63 ~ bit 56
byte 1: bit 55 ~ bit 48
byte 2: bit 47 ~ bit 40
byte 3: bit 39 ~ bit 32
byte 4: bit 31 ~ bit 24
byte 5: bit 23 ~ bit 16
byte 6: bit 15 ~ bit  8
byte 7: bit  7 ~ bit  0
```

- The parameter *`endianess`* specifies how the bytes of the `BINARY_DOUBLE` value are mapped to the bytes of the `RAW` value. In the following matrix, `rb0` to `rb7` refer to the bytes in `RAW` and `db0` to `db7` refer to the bytes in `BINARY_DOUBLE`.

	rb0	rb1	rb2	rb3	rb4	rb5	rb6	rb7
big_endian	`db0`	`db1`	`db2`	`db3`	`db4`	`db5`	`db6`	`db7`
little_endian	`db7`	`db6`	`db5`	`db4`	`db3`	`db2`	`db1`	`db0`

- When `machine_endian` is specified, the eight bytes of the `RAW` argument are copied straight across into the `BINARY_DOUBLE` return value. The effect is the same as if the user specified `big_endian` on a big-endian system or `little_endian` on a little-endian system.

Exceptions

If the `RAW` argument is less than eight bytes, a `VALUE_ERROR` exception is raised.

CAST_TO_BINARY_FLOAT function

This function casts the `RAW` binary representation of a `BINARY_FLOAT` value into a `BINARY_FLOAT` value.

Syntax

```
UTL_RAW.CAST_TO_BINARY_FLOAT (
   r          IN RAW
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN BINARY_FLOAT;
```

Parameters

Table 11–11 CAST_TO_BINARY_FLOAT function parameters

Parameter	Description
`r`	The `RAW` binary representation of a `BINARY_FLOAT` value.
`endianess`	A `BINARY_INTEGER` value indicating the endianess. The function recognizes the defined constants `big_endian`, `little_endian`, and `machine_endian`. The default is `big_endian`.

Return value

The `BINARY_FLOAT` value, or `NULL` if the input is `NULL`.

Usage notes

- If the `RAW` argument is more than four bytes, only the first four bytes are used and the rest of the bytes are ignored. If the result is -0, +0 is returned. If the result is `NaN`, the value `BINARY_FLOAT_NAN` is returned.
- A four-byte `BINARY_FLOAT` value maps to the IEEE 754 single-precision format as follows:

  ```
  byte 0: bit 31 ~ bit 24
  byte 1: bit 23 ~ bit 16
  byte 2: bit 15 ~ bit  8
  byte 3: bit 7 ~  bit  0
  ```

- The parameter *`endianess`* specifies how the bytes of the `BINARY_FLOAT` value are mapped to the bytes of the `RAW` value. In the following matrix, `rb0` to `rb3` refer to the bytes in `RAW` and `fb0` to `fb3` refer to the bytes in `BINARY_FLOAT`.

	rb0	**rb1**	**rb2**	**rb3**
big_endian	`fbo`	`fb1`	`fb2`	`fb3`
little_endian	`fb3`	`fb2`	`fb1`	`fb0`

- When `machine_endian` is specified, the four bytes of the `RAW` argument are copied straight across into the `BINARY_FLOAT` return value. The effect is the same as if the user specified `big_endian` on a big-endian system or `little_endian` on a little-endian system.

Exceptions

If the `RAW` argument is less than four bytes, a `VALUE_ERROR` exception is raised.

CAST_TO_BINARY_INTEGER function

This function casts the `RAW` binary representation of a `BINARY_INTEGER` value into a `BINARY_INTEGER` value.

Syntax

```
UTL_RAW.CAST_TO_BINARY_INTEGER (
   r          IN RAW
   endianess  IN BINARY_INTEGER DEFAULT 1)
RETURN BINARY_INTEGER;
```

Parameters

Table 11–12   CAST_TO_BINARY_INTEGER function parameters

Parameter	Description
r	The `RAW` binary representation of a `BINARY_INTEGER` value.
endianess	A `BINARY_INTEGER` value indicating the endianess. The function recognizes the defined constants `big_endian`, `little_endian`, and `machine_endian`. The default is `big_endian`.

Return value

The `BINARY_INTEGER` value, or `NULL` if the input is `NULL`.

CAST_TO_NUMBER function

This function casts the `RAW` binary representation of a `NUMBER` value into a `NUMBER` value.

Syntax

```
UTL_RAW.CAST_TO_NUMBER (
   r  IN RAW)
 RETURN NUMBER;
```

Parameters

Table 11–13   CAST_TO_NUMBER function parameters

Parameter	Description
`r`	The `RAW` binary representation of a `NUMBER` value.

Return value

The `NUMBER` value, or `NULL` if the input is `NULL`.

CAST_TO_NVARCHAR2 function

This function converts a `RAW` value represented using some number of data bytes into an `NVARCHAR2` value with that number of data bytes.

> **Note:** When casting to `NVARCHAR2`, the current Globalization Support character set is used for the characters within that `NVARCHAR2` value.

Syntax

```
UTL_RAW.CAST_TO_NVARCHAR2 (
   r IN RAW)
RETURN NVARCHAR2;
```

Parameters

Table 11–14 CAST_TO_NVARCHAR2 function parameters

Parameter	Description
r	`RAW` value, without leading length field, to be changed to an `NVARCHAR2` value.

Return value

Contains the data converted from the input `RAW` value, or `NULL` if the input is `NULL`.

CAST_TO_RAW function

This function converts a `VARCHAR2` value represented using some number of data bytes into a `RAW` value with that number of data bytes. The data itself is not modified in any way, but its data type is recast to a `RAW` data type.

Syntax

```
UTL_RAW.CAST_TO_RAW (
   c  IN VARCHAR2)
RETURN RAW;
```

Parameters

Table 11–15 CAST_TO_RAW function parameters

Parameter	Description
c	`VARCHAR2` value to be changed to a `RAW` value.

Return values

Contains the data converted from the input `VARCHAR2` value, with the same byte-length as the input value but without a leading length field. Or contains `NULL` if the input is `NULL`.

CAST_TO_VARCHAR2 function

This function converts a `RAW` value represented using some number of data bytes into a `VARCHAR2` value with that number of data bytes.

> **Note:** When casting to `VARCHAR2`, the current Globalization Support character set is used for the characters within that `VARCHAR2` value.

Syntax

```
UTL_RAW.CAST_TO_VARCHAR2 (
   r IN RAW)
RETURN VARCHAR2;
```

Parameters

Table 11–16 CAST_TO_VARCHAR2 function parameters

Parameter	Description
r	`RAW` value, without leading length field, to be changed to a `VARCHAR2` value.

Return value

Contains the data converted from the input `RAW` value, or `NULL` if the input is `NULL`.

COMPARE function

This function compares two `RAW` values. If they differ in length, then the shorter is extended on the right according to the optional `pad` parameter.

Syntax

```
UTL_RAW.COMPARE (
   r1  IN RAW,
   r2  IN RAW,
  [pad IN RAW DEFAULT NULL])
  RETURN NUMBER;
```

Parameters

Table 11–17 COMPARE function parameters

Parameter	Description
r1	First `RAW` value to be compared. It can be `NULL` or zero-length.
r2	Second `RAW` value to be compared. It can be `NULL` or zero-length.
pad	Byte to extend whichever of the input values is shorter. This is optional. The default is `x'00'`.

Return value

A `NUMBER` value that equals the position number (numbered from 1) of the first mismatched byte when comparing the two input values, or 0 if the input values are identical or both `NULL`.

CONCAT function

This function concatenates up to 12 `RAW` values into a single `RAW` value. If the concatenated size exceeds 32 KB, an error is returned.

Syntax

```
UTL_RAW.CONCAT (
   r1  IN RAW DEFAULT NULL,
   r2  IN RAW DEFAULT NULL,
   r3  IN RAW DEFAULT NULL,
   r4  IN RAW DEFAULT NULL,
   r5  IN RAW DEFAULT NULL,
   r6  IN RAW DEFAULT NULL,
   r7  IN RAW DEFAULT NULL,
   r8  IN RAW DEFAULT NULL,
   r9  IN RAW DEFAULT NULL,
   r10 IN RAW DEFAULT NULL,
   r11 IN RAW DEFAULT NULL,
   r12 IN RAW DEFAULT NULL)
  RETURN RAW;
```

Parameters

Items *`r1...r12`* are the `RAW` items to concatenate.

Return value

A `RAW` value consisting of the concatenated input values.

Exceptions

There is an error if the sum of the lengths of the inputs exceeds the maximum allowable length for a `RAW` value, which is 32767 bytes.

CONVERT function

This function converts a `RAW` value from one character set to another and returns the resulting `RAW` value.

Both character sets must be supported character sets defined to the database.

Syntax

```
UTL_RAW.CONVERT (
   r            IN RAW,
   to_charset   IN VARCHAR2,
   from_charset IN VARCHAR2)
  RETURN RAW;
```

Parameters

Table 11–18 CONVERT function parameters

Parameter	Description
`r`	`RAW` byte-string to be converted.
`to_charset`	Name of Globalization Support character set to which the input value is converted.
`from_charset`	Name of Globalization Support character set from which the input value is converted.

Return value

Contains the converted byte-string according to the specified character set.

Exceptions

`VALUE_ERROR` occurs under any of the following circumstances:

- If the input byte-string is missing, `NULL`, or zero-length.
- If *`from_charset`* or *`to_charset`* is missing, `NULL`, or zero-length.
- If *`from_charset`* or *`to_charset`* is invalid or unsupported.

COPIES function

This function returns a specified number of copies of a specified `RAW` value, concatenated.

Syntax

```
UTL_RAW.COPIES (
   r IN RAW,
   n IN NUMBER)
  RETURN RAW;
```

Parameters

Table 11–19 COPIES function parameters

Parameters	Description
`r`	`RAW` value to be copied.
`n`	Number of times to copy the `RAW` value. This must be a positive value.

Return value

The `RAW` value copied the specified number of times and concatenated.

Exceptions

`VALUE_ERROR` occurs under any of the following circumstances:

- If the value to be copied is missing, `NULL`, or zero-length.
- If the number of times to copy the value is less than or equal to 0.
- If the length of the result exceeds the maximum allowable length for a `RAW` value, which is 32767 bytes.

LENGTH function

This function returns the length in bytes of a `RAW` value.

Syntax

```
UTL_RAW.LENGTH (
   r  IN RAW)
RETURN NUMBER;
```

Parameters

Table 11–20 LENGTH function parameters

Parameter	Description
`r`	The `RAW` byte-stream to be measured.

Return value

A `NUMBER` value indicating the length of the `RAW` value, in bytes.

OVERLAY function

This function overlays the specified portion of a target `RAW` value with an overlay `RAW`, starting from a specified byte position and proceeding for a specified number of bytes.

Syntax

```
UTL_RAW.OVERLAY (
   overlay_str IN RAW,
   target      IN RAW,
  [pos         IN BINARY_INTEGER DEFAULT 1,
   len         IN BINARY_INTEGER DEFAULT NULL,
   pad         IN RAW            DEFAULT NULL])
  RETURN RAW;
```

Parameters

Table 11–21 OVERLAY function parameters

Parameters	Description
overlay_str	Byte-string used to overlay target.
target	Target byte-string to be overlaid.
pos	Byte position in target at which to start overlay (numbered from 1). This is optional. The default is 1.
len	The number of bytes to overlay. This is optional. The default is the length of *overlay_str*.
pad	Pad byte used when `len` exceeds *overlay_str* length or *pos* exceeds *target* length. This is optional. The default is `x'00'`.

Return value

Contains the `RAW` target byte value overlaid as specified.

Usage notes

If *overlay_str* has less than *len* bytes, then it is extended to *len* bytes using the *pad* byte. If *overlay_str* exceeds *len* bytes, then the extra bytes in *overlay_str* are ignored. If *len* bytes beginning at position *pos* of `target` exceed the length of *target*, then *target* is extended to contain the entire length of *overlay_str*.

`If` *len* is specified, it must be greater than or equal to 0. If *pos* is specified, it must be greater than or equal to 1. If *pos* exceeds the length of *target*, then *target* is padded with *pad* bytes to position *pos*, and *target* is further extended with *overlay_str* bytes.

Exceptions

`VALUE_ERROR` occurs under any of the following circumstances:

- If *overlay_str* is `NULL` or zero-length.
- If *target* is missing or undefined.
- If the length of *target* exceeds the maximum length for a `RAW` value, 32767 bytes.
- If *len* is less than 0.
- If *pos* is less than or equal to 0.

REVERSE function

This function reverses a `RAW` byte-sequence from end to end. For example, `x'0102F3'` would be reversed to `x'F30201'`, and `'xyz'` would be reversed to `'zyx'`. The result length is the same as the input length.

Syntax

```
UTL_RAW.REVERSE (
   r IN RAW)
  RETURN RAW;
```

Parameters

Table 11–22 REVERSE function parameters

Parameter	Description
r	`RAW` value to reverse.

Return value

Contains the `RAW` value that is the reverse of the input value.

Exceptions

`VALUE_ERROR` occurs if the input value is `NULL` or zero-length.

SUBSTR function

This function returns a substring of a `RAW` value for a specified number of bytes and starting position.

Syntax

```
UTL_RAW.SUBSTR (
   r   IN RAW,
   pos IN BINARY_INTEGER,
  [len IN BINARY_INTEGER DEFAULT NULL])
  RETURN RAW;
```

Parameters

Table 11–23 SUBSTR function parameters

Parameter	Description
r	The `RAW` byte-string from which the substring is extracted.
pos	The byte position at which to begin extraction, either counting forward from the beginning of the input byte-string (positive value) or backward from the end (negative value).
len	The number of bytes, beginning at *pos* and proceeding toward the end of the byte string, to extract. This is optional. The default is to the end of the `RAW` byte-string.

Return value

Contains the `RAW` substring beginning at position *pos* for *len* bytes, or `NULL` if the input is `NULL`.

Usage notes

If *pos* is positive, `SUBSTR` counts from the beginning of the `RAW` byte-string to find the first byte. If *pos* is negative, `SUBSTR` counts backward from the end of the `RAW` byte-string. The value of *pos* cannot equal 0.

A specified value of *len* must be positive. If *len* is omitted, `SUBSTR` returns all bytes to the end of the `RAW` byte-string.

Exceptions

`VALUE_ERROR` occurs under any of the following circumstances:

- If *pos* equals 0 or is greater than the length of `r`.
- If *len* is less than or equal to 0.
- If *len* is greater than (length of *r*) minus (*pos*-1).

Examples

Example 1

This example counts backward 15 bytes from the end of the input `RAW` value for its starting position, then takes a substring of five bytes starting at that point.

```
declare
  sr raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  r := UTL_RAW.SUBSTR(sr, -15, 5);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows:

```
source raw: 1236567812125612344434341234567890ABAA1234
return raw: 5612344434

PL/SQL procedure successfully completed.
```

Here the input and output are presented, for purposes of this discussion, in a way that gives a clearer indication of the functionality:

```
source raw: 12 36 56 78 12 12 56 12 34 44 34 34 12 34 56 78 90 AB AA 12 34
return raw: 56 12 34 44 34
```

The substring starts at the 15th byte from the end.

Example 2

This example has the same input `RAW` value and starting point as the preceding example, but because `len` is not specified the substring is taken from the starting point to the end of the input.

```
declare
  sr raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  r := UTL_RAW.SUBSTR(sr, -15);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

Here is the result:

```
source raw: 1236567812125612344434341234567890ABAA1234
return raw: 5612344434341234567890ABAA1234
```

Here the input and output are presented, for purposes of this discussion, in a way that gives a clearer indication of the functionality:

```
source raw: 12 36 56 78 12 12 56 12 34 44 34 34 12 34 56 78 90 AB AA 12 34
return raw: 56 12 34 44 34 34 12 34 56 78 90 AB AA 12 34
```

TRANSLATE function

This function performs a byte-by-byte translation of a `RAW` value, given an input set of bytes, a set of bytes to search for and translate from in the input bytes, and a set of corresponding bytes to translate to. Whenever a byte in the specified *`from_set`* is found in the input `RAW` value, it is translated to the corresponding byte in the *`to_set`* for the output `RAW` value, or it is simply not included in the output `RAW` value if there is no corresponding byte in *`to_set`*. Any bytes in the input `RAW` value that do not appear in *`from_set`* are simply copied as-is to the output `RAW` value.

Syntax

```
UTL_RAW.TRANSLATE (
   r        IN RAW,
   from_set IN RAW,
   to_set   IN RAW)
  RETURN RAW;
```

> **Note:** Be aware that *`to_set`* and *`from_set`* are reversed in the calling sequence compared to `TRANSLITERATE`.

Parameters

Table 11–24 TRANSLATE function parameters

Parameter	Description
`r`	`RAW` source byte-string whose bytes are to be translated, as applicable.
`from_set`	`RAW` byte-codes that are searched for in the source byte-string. Where found, they are translated in the result.
`to_set`	`RAW` byte-codes to translate to. Where a *`from_set`* byte is found in the source byte-string, it is translated in the result to the corresponding *`to_set`* byte, as applicable.

Return value

A `RAW` value with the translated byte-string.

Usage notes

- If *`to_set`* is shorter than *`from_set`*, the extra *`from_set`* bytes have no corresponding translation bytes. Bytes from the input `RAW` value that match any such *`from_set`* bytes are not translated or included in the result. They are effectively translated to `NULL`.
- If *`to_set`* is longer than *`from_set`*, the extra *`to_set`* bytes are ignored.
- If a byte value is repeated in *`from_set`*, the repeated occurrence is ignored.

Note: Differences from `TRANSLITERATE`:

- The *`from_set`* parameter comes before the *`to_set`* parameter in the calling sequence.
- Bytes from the source byte-string that appear in *`from_set`* but have no corresponding values in *`to_set`* are not translated or included in the result.
- The resulting `RAW` value may be shorter than the input `RAW` value.

Note that `TRANSLATE` and `TRANSLITERATE` only differ in functionality when *`to_set`* has fewer bytes than *`from_set`*.

Exceptions

`VALUE_ERROR` occurs if the source byte string, *`from_set`*, or *`to_set`* is `NULL` or zero-length.

Examples

Example 1

In this example, `from_set` is `x'12AA34'` and `to_set` is `x'CD'`. Wherever `'12'` appears in the input `RAW` value it will be replaced by `'CD'` in the result. Wherever `'AA'` or `'34'` appears in the input `RAW` value, because there are no corresponding bytes in *`to_set`*, those bytes will not be included in the result (effectively translated to `NULL`).

You can compare this to "Examples" on page 11-37 in the `TRANSLITERATE` section to see how the functions differ.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  from_set := hextoraw('12AA34');
  to_set   := hextoraw('CD');
  dbms_output.put_line('from_set:   ' || from_set);
  dbms_output.put_line('to_set:     ' || to_set);
  r := UTL_RAW.TRANSLATE(sr, from_set, to_set);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows:

```
from_set:   12AA34
to_set:     CD
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDCD56CD44CD567890ABCD

PL/SQL procedure successfully completed.
```

Here the inputs and output are presented, for purposes of this discussion, in a way that gives a clearer indication of the functionality:

```
from_set:   12  AA 34
to_set:     CD
source raw: 12 365678 12 12 56 12 34 44 34 34 12 34 567890AB AA 12 34
return raw: CD 365678 CD CD 56 CD    44       CD    567890AB    CD
```

Example 2

In this example, the *`from_set`* is `x'12AA12'` and the *`to_set`* is `x'CDABEF'`. Wherever `'12'` appears in the input RAW it will be replaced by `'CD'` in the result. Wherever `'AA'` appears in the input it will be replaced by `'AB'` in the result. The second `'12'` in *`from_set`* is ignored, and therefore the corresponding byte in *`to_set`* is ignored as well.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  from_set := hextoraw('12AA12');
  to_set   := hextoraw('CDABEF');
  dbms_output.put_line('from_set:   ' || from_set);
  dbms_output.put_line('to_set:     ' || to_set);
  r := UTL_RAW.TRANSLATE(sr, from_set, to_set);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows. Note this is the same behavior as for `TRANSLITERATE` with the same input `RAW`, *`from_set`*, and *`to_set`*, as shown in "Examples" on page 11-37.

```
from_set:   12AA12
to_set:     CDABEF
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDCD56CD34443434CD34567890ABABCD34

PL/SQL procedure successfully completed.
```

TRANSLITERATE function

This function performs a byte-by-byte transliteration of a `RAW` value, given an input set of bytes, a set of bytes to search for and convert from in the input bytes, and a set of corresponding bytes to convert to. Whenever a byte in the specified *`from_set`* is found in the input `RAW` value, it is converted to the corresponding byte in the *`to_set`* for the output `RAW` value, or it is converted to the specified "padding" byte if there is no corresponding byte in *`to_set`*. Any bytes in the input `RAW` value that do not appear in *`from_set`* are copied as-is to the output `RAW` value.

Syntax

```
UTL_RAW.TRANSLITERATE (
   r        IN RAW,
   to_set   IN RAW DEFAULT NULL,
   from_set IN RAW DEFAULT NULL,
   pad      IN RAW DEFAULT NULL)
  RETURN RAW;
```

Note: Be aware that *`to_set`* and *`from_set`* are reversed in the calling sequence compared to `TRANSLATE`.

Parameters

Table 11–25 TRANSLITERATE function parameters

Parameter	Description
`r`	`RAW` source byte-string whose bytes are to be converted, as applicable.
`to_set`	`RAW` byte-codes to convert to. Where a *`from_set`* byte is found in the source byte-string, it is converted in the result to the corresponding *`to_set`* byte, as applicable. This defaults to a `NULL` string effectively extended with *`pad`* to the length of *`from_set`*, as necessary.
`from_set`	`RAW` byte-codes that are searched for in the source byte-string. Where found, they are converted in the result. The default is `x'00'` through `x'FF'`, which results in all bytes in the source byte string being converted in the result.
`pad`	A "padding" byte that is used as the conversion value for any byte in the source byte-string for which there is a matching byte in *`from_set`* that does not have a corresponding byte in *`to_set`*. The default is `x'00'`.

Return value

A `RAW` value with the converted byte-string.

Usage notes

- If *`to_set`* is shorter than *`from_set`*, the extra *`from_set`* bytes have no corresponding conversion bytes. Bytes from the input `RAW` value that match any such *`from_set`* bytes are converted in the result to the *`pad`* byte instead.
- If *`to_set`* is longer than *`from_set`*, the extra *`to_set`* bytes are ignored.
- If a byte value is repeated in *`from_set`*, the repeated occurrence is ignored.

Note: Differences from `TRANSLATE`:

- The *`to_set`* parameter comes before the *`from_set`* parameter in the calling sequence.
- Bytes from the source byte-string that appear in *`from_set`* but have no corresponding values in *`to_set`* are replaced by *`pad`* in the result.
- The resulting `RAW` value always has the same length as the input `RAW` value.

Note that `TRANSLATE` and `TRANSLITERATE` only differ in functionality when *`to_set`* has fewer bytes than *`from_set`*.

Exceptions

`VALUE_ERROR` occurs if the source byte-string is `NULL` or zero-length.

Examples

Example 1

In this example, the *`from_set`* is `x'12AA34'` and the *`to_set`* is `x'CD'`. Wherever `'12'` appears in the input `RAW` value it will be replaced by `'CD'` in the result. Wherever `'AA'` or `'34'` appears in the input `RAW` value, because there are no corresponding bytes in *`to_set`*, those bytes will replaced by the *`pad`* byte, which is not specified and therefore defaults to `x'00'`.

You can compare this to "Examples" on page 11-34 in the `TRANSLATE` section to see how the functions differ.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  from_set := hextoraw('12AA34');
  to_set   := hextoraw('CD');
  dbms_output.put_line('from_set:   ' || from_set);
  dbms_output.put_line('to_set:     ' || to_set);
  r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows.

```
from_set:   12AA34
to_set:     CD
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDCD56CD00440000CD00567890AB00CD00

PL/SQL procedure successfully completed.
```

The inputs and output are presented in the following, for purposes of this discussion, in a way that gives a clearer indication of the functionality.

```
from_set:   12  AA 34
to_set:     CD
source raw: 12 365678 12 12 56 12 34 44 34 34 12 34 567890AB AA 12 34
return raw: CD 365678 CD CD 56 CD 00 44 00 00 CD 00 567890AB 00 CD 00
```

Example 2

This is the same as the preceding example, except *pad* is specified to be `x'FF'`.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  pad raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  from_set := hextoraw('12AA34');
  to_set   := hextoraw('CD');
  pad      := hextoraw('FF');
  dbms_output.put_line('from_set:   ' || from_set);
  dbms_output.put_line('to_set:     ' || to_set);
  r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set, pad);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows. `'AA'` and `'34'` are replaced by `'FF'` instead of `'00'`.

```
from_set:   12AA34
to_set:     CD
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDCD56CDFF44FFFFCDFF567890ABFFCDFF

PL/SQL procedure successfully completed.
```

Example 3

In this example, the *from_set* is `x'12AA12'` and the *to_set* is `x'CDABEF'`. Wherever `'12'` appears in the input `RAW` value it will be replaced by `'CD'` in the result. Wherever `'AA'` appears in the input it will be replaced by `'AB'` in the result. The second `'12'` in *from_set* is ignored, and therefore the corresponding byte in *to_set* is ignored as well.

```
declare
  sr raw(32767);
  from_set raw(32767);
  to_set raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  from_set := hextoraw('12AA12');
  to_set   := hextoraw('CDABEF');
  dbms_output.put_line('from_set:   ' || from_set);
  dbms_output.put_line('to_set:     ' || to_set);
  r := UTL_RAW.TRANSLITERATE(sr, to_set, from_set);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows. Note this is the same behavior as for `TRANSLATE` with the same input `RAW`, *`from_set`*, and *`to_set`*, as shown in "Examples" on page 11-34.

```
from_set:   12AA12
to_set:     CDABEF
source raw: 1236567812125612344434341234567890ABAA1234
return raw: CD365678CDCD56CD34443434CD34567890ABABCD34

PL/SQL procedure successfully completed.
```

Example 4

In this example, *`from_set`* and *`to_set`* are not specified.

```
declare
  sr raw(32767);
  r raw(32767);
begin
  sr       := hextoraw('1236567812125612344434341234567890ABAA1234');
  r := UTL_RAW.TRANSLITERATE(sr);
  dbms_output.put_line('source raw: ' || sr);
  dbms_output.put_line('return raw: ' || r);
end;
/
```

The result is as follows. According to the *`from_set`* and *`to_set`* defaults, all bytes are replaced by `x'00'`.

```
source raw: 1236567812125612344434341234567890ABAA1234
return raw: 000000000000000000000000000000000000000000

PL/SQL procedure successfully completed.
```

XRANGE function

This function returns a `RAW` value containing the succession of one-byte encodings beginning and ending with the specified byte-codes. The specified byte-codes must be single-byte `RAW` values. If the *`start_byte`* value is greater than the *`end_byte`* value, the succession of resulting bytes begins with *`start_byte`*, wraps through `x'FF'` back to `x'00'`, then ends at *`end_byte`*.

Syntax

```
UTL_RAW.XRANGE (
   start_byte IN RAW DEFAULT NULL,
   end_byte   IN RAW DEFAULT NULL)
  RETURN RAW;
```

Parameters

Table 11–26 XRANGE function parameters

Parameters	Description
`start_byte`	Beginning byte-code value for resulting sequence. The default is `x'00'`.
`end_byte`	Ending byte-code value for resulting sequence. The default is `x'FF'`.

Return value

`RAW` value containing the succession of one-byte encodings.

Examples

The following three examples show the results where *`start_byte`* is less than *`end_byte`*, *`start_byte`* is greater than *`end_byte`*, and default values are used.

```
Command> declare
       >    r raw(32767);
       >    s raw(32767);
       >    e raw(32767);
       > begin
       >    s := hextoraw('1');
       >    e := hextoraw('A');
       >    r := utl_raw.xrange(s,e);
       >    dbms_output.put_line(r);
       > end;
       > /
0102030405060708090A

PL/SQL procedure successfully completed.

Command> declare
       >    r raw(32767);
       >    s raw(32767);
       >    e raw(32767);
       > begin
       >    s := hextoraw('EE');
       >    e := hextoraw('A');
       >    r := utl_raw.xrange(s,e);
       >    dbms_output.put_line(r);
       > end;
       > /
EEEFF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF000102030405060708090A

PL/SQL procedure successfully completed.

Command> declare
       >    r raw(32767);
       > begin
       >    r := utl_raw.xrange();
       >    dbms_output.put_line(r);
       > end;
       > /
000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F2021222324252627
28292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F404142434445464748494A4B4C4D4E4F
505152535455565758595A5B5C5D5E5F606162636465666768696A6B6C6D6E6F7071727374757677
78797A7B7C7D7E7F808182838485868788898A8B8C8D8E8F909192939495969798999A9B9C9D9E9F
A0A1A2A3A4A5A6A7A8A9AAABACADAEAFB0B1B2B3B4B5B6B7B8B9BABBBCBDBEBFC0C1C2C3C4C5C6C7
C8C9CACBCCCDCECFD0D1D2D3D4D5D6D7D8D9DADBDCDDDEDFE0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF

PL/SQL procedure successfully completed.
```

12

UTL_RECOMP

The `UTL_RECOMP` package recompiles invalid PL/SQL modules, invalid views, index types, and operators in a database.

This chapter contains the following topics:

- Using UTL_RECOMP
  - Overview
  - Operational notes
  - Examples
- Summary of UTL_RECOMP subprograms

Using UTL_RECOMP

- Overview
- Operational notes
- Examples

Overview

`UTL_RECOMP` is particularly useful after a major-version upgrade that typically invalidates all PL/SQL objects. Although invalid objects are recompiled automatically on use, it is useful to run this before operation because this will either eliminate or minimize subsequent latencies due to on-demand automatic recompilation at runtime.

Operational notes

- This package must be run using `ttIsql`.
- To use this package, you must be the instance administrator and run it as `SYS.UTL_RECOMP`.
- This package expects the following packages to have been created with `VALID` status:
  - `STANDARD` (`standard.sql`)
  - `DBMS_STANDARD` (`dbmsstdx.sql`)
  - `DBMS_RANDOM` (`dbmsrand.sql`)
- There should be no other DDL on the database while running entries in this package. Not following this recommendation may lead to deadlocks.
- Because TimesTen does not support `DBMS_SCHEDULER`, the number of recompile threads to run in parallel is always 1, regardless of what the user specifies. Therefore, there is no effective difference between `RECOMP_PARALLEL` and `RECOMP_SERIAL` in TimesTen.

Examples

Recompile all objects sequentially:

```
Command> EXECUTE SYS.UTL_RECOMP.RECOMP_SERIAL();
```

Recompile objects in schema `SCOTT` sequentially:

```
Command> EXECUTE SYS.UTL_RECOMP.RECOMP_SERIAL('SCOTT');
```

Summary of UTL_RECOMP subprograms

Table 12–1 UTL_RECOMP Package Subprograms

Subprogram	Description
RECOMP_PARALLEL procedure	Recompiles invalid objects in a given schema, or all invalid objects in the database, in parallel. As noted earlier, in TimesTen the number of recompile threads to run in parallel is always 1, regardless of what the user specifies. Therefore, there is no effective difference between `RECOMP_PARALLEL` and `RECOMP_SERIAL` in TimesTen.
RECOMP_SERIAL procedure	Recompiles invalid objects in a given schema or all invalid objects in the database.

Notes:

- The `PLS_INTEGER` and `BINARY_INTEGER` data types are identical. This document uses `BINARY_INTEGER` to indicate data types in reference information (such as for table types, record types, subprogram parameters, or subprogram return values), but may use either in discussion and examples.
- The `INTEGER` and `NUMBER(38)` data types are also identical. This document uses `INTEGER` throughout.

RECOMP_PARALLEL procedure

This procedure uses the information exposed in the `DBA_Dependencies` view to recompile invalid objects in the database, or in a given schema, in parallel.

In TimesTen, the *`threads`* value is always 1 regardless of how it is set. As a result, there is no effective difference between `RECOMP_PARALLEL` and `RECOMP_SERIAL`.

Syntax

```
UTL_RECOMP.RECOMP_PARALLEL(
   threads  IN   BINARY_INTEGER DEFAULT NULL,
   schema   IN   VARCHAR2    DEFAULT NULL,
   flags    IN   BINARY_INTEGER DEFAULT 0);
```

Parameters

Table 12–2 RECOMP_PARALLEL procedure parameters

Parameter	Description
`threads`	The number of recompile threads to run in parallel. If `NULL`, use the value of '`job_queue_processes`'. In TimesTen, *`threads`* is always 1.
`schema`	The schema in which to recompile invalid objects. If `NULL`, all invalid objects in the database are recompiled.
`flags`	Flag values are intended for internal testing and diagnosability only.

RECOMP_SERIAL procedure

This procedure recompiles invalid objects in a given schema or all invalid objects in the database.

Syntax

```
UTL_RECOMP.RECOMP_SERIAL(
   schema   IN   VARCHAR2    DEFAULT NULL,
   flags    IN   BINARY_INTEGER DEFAULT 0);
```

Parameters

Table 12–3 RECOMP_SERIAL procedure parameters

Parameter	Description
schema	The schema in which to recompile invalid objects. If `NULL`, all invalid objects in the database are recompiled.
flags	Flag values are intended for internal testing and diagnosability only.

Index